UNIVERSITY OF
MANNHEIM

Edited by:
Ralf Gitzel, Markus Aleksy, Martin Schader, Chandra Krintz

4th International Conference on Principles and Practices of Programming in Java

Proceedings of the

4th International Conference on Principles and Practices of Programming in Java

Mannheim, Germany. August 30 – September 1, 2006

Edited by: Ralf Gitzel, Markus Aleksy, Martin Schader, Chandra Krintz

A Volume in the ACM International Conference Proceedings Series

Message from the Chairs

Dear conference participants,

We are very pleased and proud to present you with the proceedings of the 4th International Conference on the Principles and Practices of Programming in Java (PPPJ 2006). After a short break in 2005, it is good to see that the conference has not only kept its impetus of 2004 but actually has attracted even more submissions and has received increased interest.

This year's call for papers resulted in 47 submissions. In our rigorous review process, every submitted paper was carefully examined by at least three program committee members. Ultimately, the committee accepted 17 full papers and 7 short papers, leading to an acceptance rate of 36% for full papers. We are happy to say that the papers selected are of high quality and cover a wide range of topics of interest to the Java community.

Due to the support of three major interest groups (ACM SIGAPP, ACM SIGPLAN, and the GI Fachgruppe 2.1.4), we were able to recruit a wide spectrum of specialist reviewers. The program committee consisted of 36 members each with varying backgrounds, expertise, and research areas that covered the wide range of topics of interest to the PPPJ community. In a transparent and discussion-based process, only the best papers were chosen for the conference.

Organizing a conference such as the PPPJ 2006 is of course a team effort. And while it is our names that grace the front cover as editors, we gladly admit that we had the help of many very professional people. Thus, it is only fitting and proper that we take this opportunity to thank them. In particular, we are indebted to the international program committee and the PPPJ steering committee for helping to ensure the success of PPPJ06 and future events in the PPPJ line. We also thank Thomas Preuss for providing us with the excellent ConfMaster submissions software that has taken a great administrative burden from our shoulders. We owe thanks to Priya Nagpurkar, our submission chair, who very professionally contributed to the submission and review process. ACM SIGPLAN and SIGAPP as well as the German GI Fachgruppe 2.1.4 deserve our gratitude for their support, especially regarding the recruitment of reviewers and publicity for the conference. Finally, we thank the authors of all submitted papers, who have allowed us to present you with such an interesting program.

We hope that you find the 2006 PPPJ final program inspiring and that the conference provides you with the opportunity to interact, share ideas with, and learn from other Java researchers from around the world. We also encourage you to continue to participate in future PPPJ conferences, to increase its visibility, and to interest others in contributing to this growing community.

Ralf Gitzel

Markus Aleksy

Martin Schader

Chandra Krintz

PPPJ 2006 Organizing Committee

General Chairs

Ralf Gitzel, University of Mannheim (Germany)
Markus Aleksy, University of Mannheim (Germany)
Martin Schader, University of Mannheim (Germany)

Program Chair

Chandra Krintz, University of California, Santa Barbara (USA)

Submission Chair

Priya Nagpurkar, University of California, Santa Barbara (USA)

Steering Committee

Ralf Gitzel, University of Mannheim (Germany)
Markus Aleksy, University of Mannheim (Germany)
Martin Schader, University of Mannheim (Germany)
John Waldron , Trinity College Dublin (Ireland)
James Power, National University of Ireland (Ireland)

International Program Committee

Jose Nelson Amaral Univ. Alberta (Canada)
Matthew Arnold, IBM Research
Leonard Barolli, Fukuoka Institute of Technology (Japan)
Koen De Bosschere, Ghent University (Belgium)
Robert Cartwright, Rice University (USA)
John Cavazos, University of Edinburgh (UK)
Michal Cierniak, Google
Conrad Cunningham, University of Mississippi (USA)
Laurent Daynes, Sun Microsystems
Michael Franz, UC Irvine (USA)
David Gregg, University of Dublin (Ireland)
Gilles Grimaud, University of Sci/Tech of Lille (France)
Sam Guyer, Tufts Univeristy (USA)
Matthias Hauswirth, University of Lugano (Switzerland)
Martin Hirzel, IBM Research
Wade Holst, University of Western Ontario (Canada)
Patrick Hung, University of Ontario (Canada)
Richard Jones, University Kent (UK)
Gregory Kapfhammer, Allegheny College (USA)
Axel Korthaus, University of Mannheim (Germany)
Herbert Kuchen, Westfälische Wilhelms-Universität Münster (Germany)
Thomas Kühne, University of Tech. Darmstadt (Germany)
Brian Lewis, Intel
Yi Liu, South Dakota State University (USA)
Qusay H. Mahmoud, University of Guelph (Canada)
Brian Malloy, Clemson University (USA)

Table of Contents

Session A
JVM Tools

The Project Maxwell Assembler System

Bernd Mathiske, Doug Simon, Dave Ungar
Sun Microsystems Laboratories
16 Network Circle, Menlo Park, CA 94025, USA
{Bernd.Mathiske,Doug.Simon,David.Ungar}@sun.com

ABSTRACT

The Java^{TM} programming language is primarily used for platform-independent programming. Yet it also offers many productivity, maintainability and performance benefits for platform-specific functions, such as the generation of machine code.

We have created reliable assemblers for SPARC^{TM}, AMD64, IA32 and PowerPC which support all user mode and privileged instructions and with 64-bit mode support for all but the latter. These assemblers are generated as Java source code by our extensible assembler framework, which itself is written in the Java language. The assembler generator also produces javadoc comments that precisely specify the legal values for each operand.

Our design is based on the Klein Assembler System written in Self. Assemblers are generated from a specification, as are table-driven disassemblers and unit tests. The specifications that drive the generators are expressed as Java language objects. Thus no extra parsers are needed and developers do not need to learn any new syntax to extend the framework for additional ISAs.

Every generated assembler is tested against a preexisting assembler by comparing the output of both. Each instruction's test cases are derived from the cross product of its potential operand values. The majority of tests are positive (i.e., result in a legal instruction encoding). The framework also generates negative tests, which are expected to cause an error detection by an assembler. As with the Klein Assembler System, we have found bugs in the external assemblers as well as in ISA reference manuals.

Our framework generates tens of millions of tests. For symbolic operands, our tests include all applicable predefined constants. For integral operands, the important boundary values, such as the respective minimum, maximum, 0, 1 and -1, are tested. Full testing can take hours to run but gives us a high degree of confidence regarding correctness.

PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany.

Keywords

cross assembler, assembler generator, disassembler, automated testing, the Java language, domain-specific framework, systems programming

1. INTRODUCTION AND MOTIVATION

Even though the Java programming language is designed for platform-independent programming, many of its attractions[1] are clearly more generally applicable and thus also carry over to platform-specific tasks. For instance, popular integrated development environments (IDEs) that are written in the Java language have been extended (see e.g. [5]) to support development in languages such as C/C++, which get statically compiled to platform-specific machine code. Except for legacy program reuse, we see no reason why compilers in such an environment should not enjoy all the usual advantages attributed to developing software in the Java language (in contrast to C/C++). Furthermore, several Java virtual machines have been written in the Java language (e.g., [3], [21], [14]), including compilers from byte code to machine code.

With the contributions presented in this paper we intend to encourage and support further compiler construction research and development in Java. Our software relieves programmers of arguably the most platform-specific task of all, the correct generation of machine instructions adhering to existing general purpose instruction set architecture (ISA) specifications.

We focus on this low-level issue in clean separation from any higher level tasks such as instruction selection, instruction scheduling, addressing mode selection, register allocation, or any kind of optimization. This separation of concerns allows us to match our specifications directly and uniformly to existing documentation (reference manuals) and to exploit pre-existing textual assemblers for *systematic, comprehensive* testing. Thus our system virtually eliminates an entire class of particularly hard-to-find bugs and users gain a fundament of trust to build further compiler layers upon.

Considering different approaches for building assemblers, we encounter these categories:

[1]To name just a few: automatic memory management, generic static typing, object orientation, exception handling, excellent IDE support, large collection of standard libraries.

Stand-alone assembler programs: These take textual input and produce a binary output file. Compared to the other two variants they are relatively slow and they have long startup times. Therefore stand-alone assemblers are primarily used for static code generation. On the plus side, they typically offer the richest feature sets beyond mere instruction encoding: object file format output, segmentation and linkage directives, data and code alignment, macros and much more.

Inline assemblers: Some compilers for higher programming languages (HLLs) such as C/C++ provide direct embedding of assembly language source code in HLL source code. Typically they also have syntactic provisions to address and manipulate HLL entities in assembly code.

Assembler libraries: These are aggregations of HLL routines that emit binary assembly instructions (e.g., [9], [13]). Their features may not always be directly congruent with textual assembly language. For example, many methods in the x86 assembler library that is part of the HotSpot^{TM} Java virtual machine [18], which is written in C++, take generalized location descriptors as parameters instead of explicitly broken down operands. Further along this path, the distinction between a mere assembler and the following category becomes quite diffuse.

Code generator libraries: These integrate assembling with higher level tasks, typically instruction selection and scheduling, arranging ABI compliance, etc., or even some code optimization techniques.

We observed that only the first of the above categories of assemblers is readily available to today's Java programmers: that is, stand-alone assemblers, which can be invoked by the `System.exec()` method. This approach may be sufficient for some limited static code generation purposes, but it suffers from the lack of language integration and the administrative overhead resulting from having separate programs. Regarding dynamic code generation, the startup costs of external assembler processes are virtually prohibitive.

We are not aware of any inline assembler in any Java compiler and to our knowledge there are only very few assembler libraries in the form of Java packages[2].

According to our argument above concerning the separation of concerns, building a code generator framework would have to be an extension of, rather than an alternative to, our contributions.

In this paper we present a new assembler library (in the form of Java packages) that covers the most popular instruction sets (for any systems larger than handhelds): SPARC, PowerPC, AMD64 and IA32. In addition our library includes matching disassemblers, comprehensive assembler test programs, and an extensible framework with specification-driven generators for all parts that depend on ISA specifics. All of the above together comprise the Project Maxwell Assembler System (PMAS).

The design of the PMAS is to a large extent derived from the Klein Assembler System (KAS), which has been developed previously by some of us as part of the Klein Virtual Machine [19]. The KAS, which supports the two RISC ISAs SPARC and PowerPC, is written in Self [1], a prototype-based, object-oriented language with *dynamic* typing.

[2]We exclude Java byte code "assemblers", since they do not generate hardware intructions.

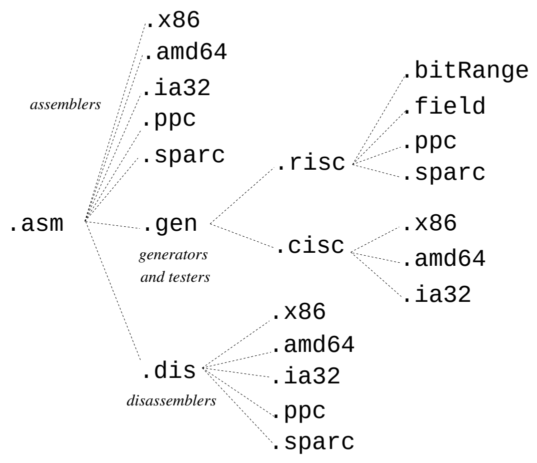


Figure 1: Overview of the PMAS packages

Providing an assembler with a programmatic interface already delivers a significant efficiency gain over a textual input based assembler as the cost of parsing is incurred during Java source compilation instead of during the execution of the assembler. In addition, we will discuss how making appropriate usage of Java's type system can also shift some of the cost of input validation to the Java source compiler.

2. OVERVIEW

Figure 1 gives an overview of the PMAS package structure.[3] The `.gen` package and its subpackages contain ISA descriptions and miscellanous generator and test programs. The `.dis` subtree implements disassemblers, reusing the `.gen` subtree. The other five direct subpackages of `.asm` contain assemblers, which have no references to the above two subtrees. Each package ending in `.x86` provides shared classes for either colocated `.amd64` and `.ia32` packages.

It is straightforward to reduce the framework to support any subset of ISAs, simply by gathering only those packages that do not have any names which pertain only to excluded ISAs.

The next section explains how to use the generated assemblers in Java programs. For each assembler there is a complementary disassembler, as described in section 4. We describe the framework that creates these assemblers and disassemblers in section 5. First we introduce the structure of our ISA representations (section 5.1), then we sketch the instruction templates (section 5.2) that are derived from them. We regard the latter as the centerpiece of our framework, as it is shared among its three main functional units, which provide the topics of the subsequent three sections: the structure of generated assembler source code (section 5.3), then the main disassembler algorithms (section 5.4) and fully au-

[3]We have split our source code into two separate IDE projects, one of which contains miscellaneous general purpose packages that are reused in other projects and that we regard as relatively basic extensions of the JDK. Here we only discuss packages unique to the PMAS.

tomated assembler and disassembler testing (section 5.5). We briefly discuss related work in section 6 and the paper concludes with notable observations and future work (section 7).

3. HOW TO USE THE ASSEMBLERS

Each assembler consists of the top level package `com.sun.max.asm` and the subpackage matching its ISA as listed in Figure 1. In addition, the package `com.sun.max.asm.x86` is shared between the AMD64 and the IA32 assembler. Hence, to use the AMD64 assembler the following packages are needed:[4] `com.sun.max.asm`, `com.sun.max.asm.amd64` and `com.sun.max.asm.x86`. None of the assemblers requires any of the packages under `.gen` and `.dis`.

To use an assembler, one starts by instantiating one of the leaf classes shown in Figure 2. The top class `Assembler` provides common methods for all assemblers, concerning e.g. label binding and output to streams or byte arrays. The generated classes in the middle contain the ISA-specific assembly routines. For ease of use, these methods are purposefully closely oriented at existing assembly reference manuals, with method names that mimic mnemonics and parameters that directly correspond to individual symbolic and integral operands.

Here is an example for AMD64 that creates a small sequence of machine code instructions (shown in Figure 3) in a Java byte array:

```
import static
    ...asm.amd64.AMD64GeneralRegister64.*;
    ...
public byte[] createInstructions() {
    long startAddress = 0x12345678L;
    AMD64Assembler asm =
        new AMD64Assembler(startAddress);

    Label loop = new Label();
    Label subroutine = new Label();
    asm.fixLabel(subroutine, 0x234L);

    asm.mov(RDX, 12, RSP.indirect());
    asm.bindLabel(loop);
    asm.call(subroutine);
    asm.sub(RDX, RAX);
    asm.cmpq(RDX, 0);
    asm.jnz(loop);

    asm.mov(20, RCX.base(), RDI.index(),
            SCALE_8, RDX);

    return asm.toByteArray();
}
```

Instead of using a byte array, assembler output can also be directed to a stream (e.g. to write to a file or into memory):

```
OutputStream stream = new ...Stream(...);
asm.output(stream);
```

The above example illustrates two different kinds of label usage. Label `loop` is bound to the instruction following the `bindLabel()` call. In contrast, label `subroutine` is bound to an absolute address. In both cases, the assembler creates PC-relative code, though, by computing the respective offset argument.[5] An explicit non-label argument can be expressed by using `int` (or sometimes `long`) values instead of labels, as in:

```
asm.call(200);
```

The variant of `call()` used here is defined in the *raw* assembler (`AMD64RawAssembler`) superclass of our assembler and it takes a "raw" int argument:

```
public void call(int rel32) { ... }
```

In contrast, the `call()` method used in the first example is defined in the *label* assembler (`AMD64LabelAssembler`), which sits between our assembler class and the raw assembler class:

```
public void call(Label label) {
  ... call(labelOffsetAsInt(label)); ...
}
```

This method builds on the raw `call()` method, as sketched in its body.

These methods, like many others, are syntactically differentiated by means of parameter overloading. This Java language feature is also leveraged to distinguish whether a register is used directly, indirectly, or in the role of a base or an index. For example, the expression `RSP.indirect()` above results in a different Java type than plain `RSP`, thus clarifying which addressing mode the given `mov` instruction must use. Similarily, `RCX.base()` specifies a register in the role of a base, etc.

If there is an argument with a relatively limited range of valid values, a matching *enum* class rather than a primitive Java type is defined as the parameter type. This is for instance the case regarding `SCALE_8` in the SIB addressing expression above. Its type is declared as follows:

```
public enum Scale ... {
  SCALE_1, SCALE_2, SCALE_4, SCALE_8;
   ...
}
```

Each RISC assembler features *synthetic* instructions according to the corresponding reference manual. For instance, one can write these statements to create some synthetic SPARC instructions [20]:

```
import static ...asm.sparc.GPR.*;

SPARC32Assembler asm = new SPARC32Assembler(...);
asm.nop();
asm.set(55, G3);
asm.inc(4, G7);
asm.retl();
 ...
```

Let's take a look at the generated source code of one of these methods:

[4]In addition, general purpose packages from MaxwellBase and the JRE are needed.

[5]In our current implementation, labels always generate PC-relative code, i.e. absolute addressing is only supported by the *raw* assemblers.

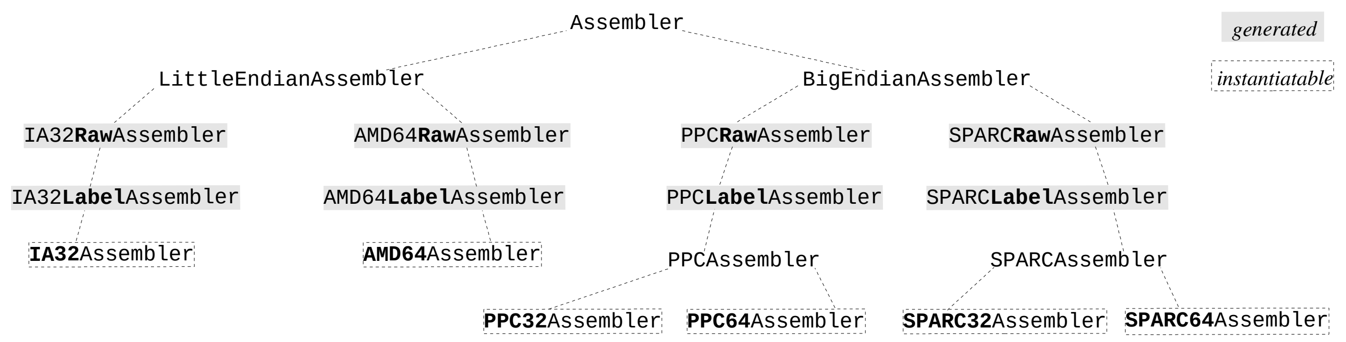


Figure 2: Assembler class hierarchy

```
/**
 * External assembler syntax:
 * <code>inc</code>  <i>simm13</i>, <i>rd</i>
 * <p>
 * Synthetic instruction equivalent to:
 * <code>{@link #add(GPR, int, GPR) add}
    (rd.value(), simm13, rd)</code>
 * <p>
 * Constraint: <code>-4096 <= simm13 &&
    simm13 <= 4095</code><br />
 *
 * @see "<a href="...</a> - Section G.3"
 */
public void inc(int simm13, GPR rd) {
  int instr = 0x80002000;
  checkConstraint(-4096 <= simm13 &&
                  simm13 <= 4095,
    "-4096␣<=␣simm13␣&&␣simm13␣<=␣4095");
  instr |= ((rd.value() & 0x1f) << 14);
  instr |= (simm13 & 0x1fff);
  instr |= ((rd.value() & 0x1f) << 25);
  emitInt(instr);
}
```

As we see here, our assembler generator creates a javadoc comment disclosing the textual external assembler syntax of each instruction and pointing out the exact place ("section G.3") in the reference manual [20] to find a detailed instruction description.

As is often the case for RISC (but not x86) assembler methods, the `int` argument above is limited to a value range of less than 32 bits, here only 13. In such cases, we resort to dynamic checking resulting in runtime exceptions if an out-of-range argument is passed. In all other situations, our assemblers are statically type-safe.

4. HOW TO USE THE DISASSEMBLERS

Since performance is less critical for disassemblers than for assemblers, our disassemblers are not generated as Java source code. Instead they are manually written programs, which on every program restart rely on the PMAS framework to generate instruction *template* tables (see 5.2). Thus using a disassembler always requires loading the respective packages under `.gen` as well.

This Java statement sequence disassembles AMD64 instructions from an input stream, given a start address for PC-relative decoding, and directs its textual output to the console:

```
AMD64Disassembler disasm =
  new AMD64Disassembler(startAddress);
BufferedInputStream stream = ...;
  new BufferedInputStream(...);
disasm.scanAndPrint(stream, System.out);
```

Applied to the instructions in the AMD64 code example at the beginning of the preceding section, this produces the output in Figure 3.

Our disassembled syntax for AMD64 (and IA32) is a blend of so-called Intel and AT&T syntax [6], with some modifications regarding address parameterization. Our address rendering mimics C/Java syntax, which we find much more intuitive.

As demonstrated by label `L1` in Figure 3, the disassembler automatically synthesizes labels for any address in the range of the disassembled code. Instructions that reference a label are printed giving both the label and its underlying raw value.

The disassemblers for SPARC and PowerPC create textual output that is virtually identical to the syntax found in the reference manuals, except for also providing label synthesis. In any case, the source code that needs to be changed to adjust any disassembler's output to one's personal liking is fairly minimal.

5. THE GENERATOR FRAMEWORK

Having described the use of our assemblers and disassemblers, we will now address how they are implemented and tested. The following subsections describe how our framework starts from the internal description of an ISA and derives an abstract representation of assembler methods called *templates*. These constitute the centerpiece of the assembler generators, the disassemblers and for fully automated testing.

5.1 Constructing an ISA Representation

We represent each ISA by a collection of instruction *descriptions* in the form of Java object arrays. Each of these specifies the exact composition of a group of closely related instructions.

The first object in every description must be a string which specifies the *external name*, i.e. the instruction mnemonic used in external assembler syntax. This string will also be used as the *internal name*, i.e., the base name for generated assembler methods, unless a second string is also given, which then defines a different internal name. For example, as the `return` instruction in the SPARC ISA clashes with a Java keyword, we gave it the internal name `return_`.

```
memory address      offset   mnemonic  arguments                 raw bytes in hex

0x0000000012345678  00       mov       rdx, [rsp + 12]           [48 8B 94 24 0C 00 00 00]
0x0000000012345680  08  L1:  call      -305419345                [E8 AF AB CB ED]
0x0000000012345685  13       sub       rdx, rax                  [48 29 C2]
0x0000000012345688  16       cmpq      rdx, 0x0                  [48 81 FA 00 00 00 00]
0x000000001234568F  23       jnz       L1: -17                   [75 EF]
0x0000000012345691  25       mov       rcx[rdi * 8 + 20], rdx    [48 89 94 F9 14 00 00 00]
```

Figure 3: Disassembled AMD64 instructions

5.1.1 RISC Instruction Descriptions

A RISC instruction has 32 bits and it is typically specified as sequences of bit fields. See for example the specification of the `casa` instruction from the SPARC reference manual [20]:

11	rd	op3	rs1	i=0	imm_asi	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

Our system follows this structure by treating RISC instructions as a sequence of bit fields that either have constant values or an associated operand type. We have the following description types at our disposal:

RiscField: describes a bit range and how it relates to an operand. This implicitly specifies an assembly method parameter. A field may also append a suffix to the external and internal names.

RiscConstant: combines a field with a predefined value, which will occupy the field's bit range in each assembled instruction. This constitutes a part of the instruction's opcode.

OperandConstraint: a predicate that constrains the legal combination of argument values an assembler method may accept.

String: a piece of external assembler syntax (e.g., a parenthesis or a comma), that will be inserted among operands, at a place that corresponds to its relative position in the description.

Here is our specification of the `casa` instruction:

```
define("casa", op(0x3), op3(0x3c), "[", _rs1,
  "]_", i(0), _imm_asi, ",", _rs2, _rd);
```

This specifies the name, the external syntax and the fields of the `casa` instruction. There are three constant fields (`RiscConstant`), `op`[6], `op3` and `i`, as well as 4 variable fields (`RiscField`), `_rs1`, `_imm_asi`, `_rs2` and `_rd`. An example using the external syntax would be:

```
casa [G3]12, I5
```

The same field may be constant in some instructions, but variable in others. When writing field definitions for an ISA, one defines the same field in two ways, once as a Java value and once as a Java method.

[6]This field is not named in the reference manual diagram.

```
public static final
 SymbolicOperandField<GPR> _rd =
  createSymbolicOperandField(GPR.SYMBOLIZER,
                             29, 25);

public static RiscConstant rd(GPR gpr) {
    return _rd.constant(gpr);
}

private static final ConstantField _op3 =
  createConstantField(24, 19);

public static RiscConstant op3(int value) {
    return _op3.constant(value);
}
...
public class ConstantField extends RiscField {
  RiscConstant constant(int value) {
      return new RiscConstant(this, value);
  }
}
```

This means that one can reference the return register field as a parameter operand (`_rd`) or as a constant field (e.g., `rd(G3)`). Field `op3` however, is always supposed to be constant, so we made only its method `public`.

The following specifies a PowerPC instruction featuring an opcode field and 6 parameter fields:

```
define("rlwinm", opcd(21), _ra, _rs,
       _sh, _mb, _me, _rc);
```

In all instruction descriptions we apply the *static import* feature of the Java language, to avoid qualifying static field and method names. This greatly improved both the ease of writing descriptions and their readability. For instance, the above would otherwise have to be written as:

```
define("rlwinm", PPCFIelds.opcd(21),
    PPCFields._ra, PPCFields._rs,
    PPCFields._sh, PPCFields._mb,
    PPCFields._me, PPCFields._rc);
```

The PowerPC ISA is particularly replete with *synthetic* instructions, the specifications of which build on other instructions [11]. To match the structure of existing documentation closely, there is a `synthesize` method that derives a synthetic instruction from a previously defined (raw or synthetic) instruction. This method interprets its instruction description arguments to replace parameters of the referenced instruction with constants or alternative parameters. For example, we can define the *rotlwi* instruction by referring to the above *rlwinm* instruction:

```
synthesize("rotlwi", "rlwinm", sh(_n),
           mb(0), me(31), _n);
```

Here, we specify a new parameter field `_n` and cause the generated assembly method to assign its respective argument to field `_sh`. The fields `_mb` and `_me` become constant with the given predefined values.

Furthermore, fields in synthetic instructions can be specified by arithmetic expressions composed of numeric constants and fields. For example, the values of the `mb` and `me` fields in the following instruction description are the result of subtraction expressions.

```
synthesize("clrlslwi", "rlwinm", sh(_n),
           mb(SUB(_b, _n)), me(SUB(31, _n)),
           _b, _n, LE(_n, _b), LT(_b, 32));
```

The repeated use of field `_n` exemplifies how one operand may contribute to the values of several fields.

5.1.2 x86 Instruction Descriptions

The number of possible instructions in x86 ISAs is about an order of magnitude larger than in the given RISC ISAs. If one tried to follow the same approach to create instruction descriptions, one would spend an enormous amount of time just writing the description listings. More importantly, our primitives to specify RISC instructions are insufficient to express instruction prefixes, suffixes, intricate mod r/m relationships, etc. Instead of a rich bit-field structure, x86 instructions tend to have a byte-wise composition determined by numerous not quite orthogonal features.

As opcode tables provide the densest, most complete, well-publicized instruction set descriptions available for x86, we decided to build our descriptions and generators around those. For an x86 ISA, the symbolic constant values of the following description object types are verbatim from opcode tables found in x86 reference manuals (e.g., [12]):

AddressingMethodCode: We allow `M` to be used in lieu of the operand code `Mv` to faithfully mirror published opcode tables in our instruction descriptions.

OperandTypeCode: e.g. `b, d, v, z`. Specifies a mnemonic suffix for the external syntax.

OperandCode: the concatenation of an addressing mode code with an operand type code, e.g. Eb, Gv, Iz, specifies explicit operands, resulting in assembler method parameters.

RegisterOperandCode: e.g. `eAX, rDX`.

GeneralRegister: e.g. `BL, AX, ECX, R10`.

SegmentRegister: e.g. `ES, DS, GS`.

StackRegister: e.g. `ST, ST_1, ST_2`.

The latter three result in implicit operands, i.e. the generated assembler methods do not represent them by parameters. Instead we append an underscore and the respective operand to the method name. For example, the external assembly instruction `add EAX, 10` becomes `add_EAX(10)` when using the generated assembler. We also generate the variant with an explicit parameter that can be used as `add(EAX, 10)`, but that is a *different* instruction, which is one byte longer in the resulting binary form. External textual assemblers typically do not provide any way to express such choices.

In addition, these object types are used to describe x86 instructions:

HexByte: an enum providing hexadecimal unsigned byte values, used to specify an opcode. Every x86 instruction has either one or two of these. In case of two, the first opcode must be `0F`.

ModRMGroup: specifies a table in which alternative additional sets of instruction description objects are located, indexed by the respective 3-bit opcode field in the mod r/m byte of each generated instruction.

ModCase: a 2-bit value to which the *mod* field of the *mod r/m* byte is then constrained.

FloatingPointOperandCode: a floating point operand not further described here.

Integer: an implicit byte operand to be appended to the instruction, typically `1`.

OperandConstraint: same as for RISC above, but much more rarely used, since almost all integral x86 operand value ranges coincide with Java primitive types.

Given these features, we can almost trivially transcribe the "One Byte Opcode Map" for IA32:

```
define(_00, "ADD", Eb, Gb);
define(_01, "ADD", Ev, Gv);
...
define(_15, "ADC", eAX, Iv);
define(_16, "PUSH", SS);
...
define(_80, GROUP_1, b,
  Eb.excludeExternalTestArgs(AL), Ib);
...
define(_CA, "RETF",
       Iw).beNotExternallyTestable();
       // gas does not support segments
...
define(_6B, "IMUL", Gv, Ev,
       Ib.externalRange(0, 0x7f));
...
```

Many description objects and the respective result value of `define` have modification methods that convey special information to the generator and the tester. In the example above we see the exclusion of a register from testing, the exclusion of an entire instruction from testing and the restriction of an integer test argument to a certain value range. These features suppress already known testing errors that are merely due to restrictions, limited capabilities, or bugs in a given external assembler.

Analogous methods to the above are available for RISC instruction descriptions. For x86, however, there are additional methods that modify generator behavior to match details of the ISA specification which are not explicit in the opcode table. This occurs for example in the "Two Byte Opcode Table" for AMD64:

```
define(_0F, _80, "JO",
  Jz).setDefaultOperandSize(BITS_64);
...
define(_0F, _C7,
  GROUP_9a).requireAddressSize(BITS_32);
```

5.2 Instruction Templates

The assembler generator, the disassembler and the assembler tester of each ISA share a common internal representation derived from instruction descriptions called instruction *templates* and a common mechanism to create these, the *template generator*.

For RISC, an instruction template is created by binding constants to all the non-parameter operands in an instruction description and by building the cross product of all possible bindings for option fields.

The template generator for x86 is far more complex. It explores the cross product of all possible values of the following instruction properties, considering them in this order: address size attribute, operand size attribute, mod case, mod r/m group, rm case, SIB index case and SIB base case. The search for valid combinations of the above is directed by indications derived from the respective instruction description objects.

For shorter instructions, a result may be found after the first few stages. For example, an instruction that does not have a mod r/m byte, as determined by examining its opcode, may have templates with different address and operand size attributes, but enumerating different mod cases, etc., is unnecessary.

There are numerous relatively difficult to describe circumstances that limit the combinatorial scope for valid instructions. In such cases, the template generator internally throws the exception `TemplateNotNeededException` in the respective description object visitor to backtrack among the above stages. For example, instructions with addressing method code `M` occurring in their description do not require consideration of any other rm cases than the normal case when exploring mod case 3. In other words, if two general registers are used directly as operands (mod case 3), then there will be no complex addressing forms involving a SIB byte and no special (rm) cases such as memory access by an immediate address.

The number of templates that can be generated for any given instruction description ranges anywhere from 1 (for most RISC instructions) to 216 (for the *xor* AMD64 instruction).

5.3 Generating Assembler Source Code

Each assembler generator writes two Java source code classes containing hundreds or thousands of assembly methods:[7] a *raw* assembler class and a *label* assembler class. As indicated in Figure 2, these generated classes are accompanied by manually written classes that implement all necessary support subroutines as e.g. output buffering, label definition and binding, and instruction length adjustments and that define symbolic operand types, such as registers, special constants, etc.

For x86, we managed to use Java enums to represent all symbolic operands. For most symbolic RISC operands, though, we had to resort to a manually created pattern that mimics enums in order to capture relatively complex interrelationships such as subsetting. For example, only every second SPARC floating point register syntactically can be "double" and only every fourth can be "quadruple".

[7]The generated AMD64 assembler (without optional 16-bit addressing) contains 8450 methods in ≈85k lines of code, half of which are comments. The totals for the SPARC assembler are 832 methods and ≈13k lines of code.

By limiting the scope of all symbolic operand constructors to their respective class we restrict symbolic operands to predefined constants and thus we *syntactically* and therefore *statically* prevent the passing of illegal arguments to assembler methods.

To represent integral values we use Java primitive types (i.e., `int`, `short`, `char`, etc) of the appropriate value size. If the range of legal values for an integral parameter does not exactly correspond to the range of legal values for the Java type then we add a constraint accordingly to the instruction description.

A generator needs to be run only once per assembler release. It also programmatically invokes a Java compiler[8] to reduce the generated source code to class files.

A generated raw assembler class contains one assembly method for every instruction template derived from the given ISA description. The corresponding label assembler class inherits all these methods and provides additional derivative methods with *label* parameters in lieu of primitive type (raw) parameters, wherever this is useful, based on the semantics of the respective instruction.

Each label assembler method translates its label arguments to raw operands and calls the corresponding underlying raw assembler method. In the case of x86, label assembler methods also support span-dependent instruction selection. The inherited top level assembler class (see Figure 2) provides reusable algorithms for label resolution and for span-dependent instruction management.

The generated code that assembles a RISC instruction shifts and masks incoming parameters into position and combines the resulting bits using the logical *or* operation.

The assembly of x86, on the other hand, is mostly organized as a sequence of bytes, with conditional statements guarding the emission of certain prefixes. Some more complex bytes such as mod r/m bytes or REX prefixes also require a certain amount of bit combining.

5.4 Implementing Disassemblers

The disassemblers also reuse the template generator, but they are entirely manually written. They have simple, almost but not quite brute force algorithms with usable but not great performance. At startup, a disassembler first creates all templates for the given ISA. When applied to an instruction stream it then tries to find templates that match its binary input. The details for this task vary between the RISC and x86 disassemblers. They are described in the following two subsections. In either case, the disassembler extracts operand values and then produces a textual output including program counter addresses, offsets, synthesized labels and raw bytes.

5.4.1 RISC Disassemblers

A SPARC or PowerPC disassembler only needs to read a 32-bit word to obtain a full bit image of any given instruction. To explain how it then finds a matching template, we use these notions:

[8]Programmatic Java compiler invocation is provided for both Sun's javac (used in NetBeans) and for IBM's Jikes compiler (used in Eclipse).

opcode mask: the combined (not necessarily contiguous) bit range of all constant fields in an instruction,

opcode: a binding of bit values to a given opcode mask,

opcode mask group: a collection of templates that share the same opcode mask,

specificity: the number of bits in an opcode mask,

specificity group: a collection of opcode mask groups with the same specificity (but different opcode bit positions).

The disassembler keeps all instruction templates sorted by specificity group. To find the template with the most specific opcode mask it will iterate over all specificity groups in order of decreasing specificity. Optionally, it can do the opposite.

During the iteration, the disassembler tries the following with each opcode mask group in the given specificity group. A logical *and* of the opcode mask group's opcode mask with the corresponding bits in the instruction yields a hypothetical opcode. Next, every template in the opcode mask group that has this opcode is regarded as a candidate. For each such candidate, the disassembler tries to disassemble the instruction's encoded arguments.

If this succeeds, we reassemble an instruction from the candidate template with these arguments. This simple trick assures that we only report decodings that match all operand constraints. Finally, if the resulting bits are the same as the ones we have originally read, we have a result.

5.4.2 x86 Disassemblers

An AMD64 and IA32 assembler must determine the instruction length on the fly, sometimes backtracking a little bit. An instruction is first scanned byte by byte, gathering potential prefixes, the first opcode, and, if present, the second opcode. The disassembler can then determine a group of templates that matches the given opcode combination, ignoring any prefixes at the moment.

The disassembler iterates over all templates in the same *opcode group*, extracts operand values and reassembles as described above in case of RISC. In short, some effort is made to exclude certain predictably unnecessary decoding attempts, but overall, the x86 disassembler algorithm uses an even more brute force approach than the RISC disassemblers.

5.5 Fully Automated Self-Testing

The same template generators used to create assemblers and disassemblers are reused once again for fully automated testing of these artifacts. The respective test generator creates an exhaustive set of test cases by iterating over a cross product of legal values for each parameter of an assembler method. For symbolic parameters, the legal values amount to the set of all predefined constants of the given symbol type. For integral parameters, if the number of legal values is greater than some plausible threshold (currently 32), only a selection of values representing all important boundary cases is used.

The above represent positive test cases, i.e., they are expected to result in valid instruction encodings. In addition, the testing framework generates negative test cases, i.e., test cases that should cause an assembler to display error behavior (e.g., return an error code or throw an exception). There

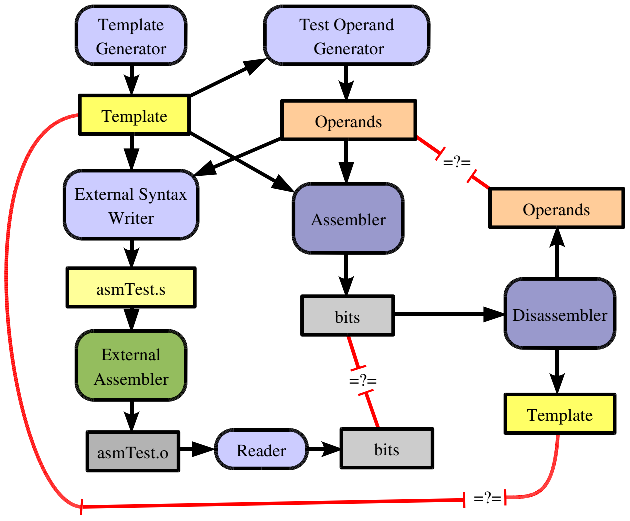


Figure 4: Testing

will be far fewer negative test cases than positive test cases as our use of Java's static typing leaves very few opportunities for specifying illegal arguments. By far most negative test cases in the ISAs implemented so far are due to RISC integral fields whose ranges of legal values are not exactly matched by a Java primitive type (e.g., int, short, char, etc).

For complete testing of an assembler and its corresponding disassembler, the template generator creates all templates for an ISA and presents them one by one to the following testing procedure.[9]

First, an assembler instance is created and the assembler's Java method that corresponds to the given template is identified by means of Java reflection. Then the test generator creates all test case operand sets for the given template.

Figure 4 illustrates the further steps taken for each operand set. The assembler method is invoked with the operand set as arguments and the identical operand set is also passed together with the template to the external syntax writer, which creates a corresponding textual assembler code line and writes it into an assembler source code file. The latter is thereupon translated by an external assembler program (e.g., gas [6]), producing an object file. By noticing certain prearranged markers in the object file, a reader utility is able to extract those bits from the object file that correspond to output of the external syntax writer into another byte array.

Now the two byte arrays are compared. Next, one of the byte arrays is passed to the disassembler, which reuses the same set of templates from above and determines, which template exactly matches this binary representation. Furthermore, the disassembler extracts operand values.

Eventually, the template and operand values determined by the disassembler are compared to the original template and operand values.

Probing all variants of even a single mnemonic may take minutes. Once a test failure has been detected, we can ar-

[9]The description is slightly simplified: the actual program does not write a new assembler source file per instruction, but accumulates those for the same template.

range for a very short restart/debug cycle by limiting testing to a subset of instruction templates. The instruction templates in question are easily identified by serial numbers, which are listed in the test output. When restarting the test program, we then constrain the range of templates to be retested by means of a command line option.

6. RELATED WORK

We have based our design on the Klein Assembler System (KAS), deriving for example the following features from it:

- specification-driven generation of assemblers, disassemblers and testers,
- instruction descriptions in the form of object arrays,
- instruction templates as the central internal representation for multiple purposes,
- most of the generator, disassembler and tester algorithms for RISC instructions,
- employment of existing external assemblers for testing.

Furthermore, we were able to copy and then simply transcode the instruction descriptions for SPARC and PowerPC. The x86 part of the PMAS has no precedent in the KAS. Whereas the general approach carried over and there is considerable reuse between the RISC and the x86 part of the framework, we had to devise different instruction descriptions, template structures, template generators and disassemblers for AMD64 and IA32.

The NJMC toolkit [16] has many similarities with our architecture. It is a specification driven framework for generating assemblers and disassemblers. Like ours, it includes mechanisms for checking the correctness of a specification internally as well as against an external assembler [7]. However, the decoders it generates are more efficient and extensible.[10] Also, it produces C source code, uses a special language for the specifications (SLED [17]) and is implemented in ML. This use of three different languages makes using, extending and modifying the toolkit harder in the context of a Java based compilation system. In contrast, the PMAS uses Java as the single language for all its purposes.

Other specification language based assembler generators are also described in [22], [4] and similar publications about cross assemblers.

There are several extremely fast code generator and assembler frameworks written for and implemented in C/C++ which are specially apt for dynamic code generation.

GNU lightning [9] provides an abstract RISC-like instruction set interface. This makes assembly code written to this interface highly portable while trading off complete control over the native instructions that are emitted.

CCG [15] is a combination of preprocessor and runtime assembler that allows code generation to be embedded in arbitrary C programs and requires no compiler-specific extensions (such as inline asm statements or the various assembler-related extensions implemented by gcc). It gives the programmer complete control over what instructions are emitted.

One can find many more code generator frameworks (e.g., [8]) for C/C++.

[10]This is something we plan to remedy as described in section 7.

7. OBSERVATIONS AND FUTURE WORK

Conventionally, assemblers that run on one hardware architecture and generate code for another are categorized as *cross assemblers* and those that don't are not. Interestingly, this categorization is no longer static, i.e. determined at assembler build time, when it comes to assemblers written in a platform-independent language such as Java. Whether they cross-assemble is merely a dynamic artifact of invoking them on different platforms. On the other hand, one could argue that they run on a virtual instruction set, Java byte codes, and are therefore *always* cross-assembling.

Developing in the Java 5 language [10], we found that the features (generics, enums, static imports, varargs, annotations, etc.) introduced in this release of the language contributed substantially to our design, especially to support static type-safety of assembler applications.

The use of static typing by the Java compiler to prevent expressing illegal operand values greatly reduces the number of negative tests (e.g. $\approx$6000 for AMD64). Most are derived from RISC instruction operands that cannot be modelled precisely by a Java primitive type (e.g. int, short, char, etc).

We first created a minimal framework that would only cover very few instructions but contained all parts necessary to run our instruction testing suite as described in section 5.5. Thus we were able to catch numerous bugs, resulting from faulty instruction descriptions, missing features in our framework and various mistakes, early on. Then we expanded the number of instruction descriptions and added functionality as needed.

The effort required to develop the assembler generators shrank with each successive ISA. The CISC ISAs (IA32 and AMD64) took about 3 engineer months to complete. A large portion of this time can be attributed to the development and extension of the general framework as these were the first ISAs we implemented. The SPARC and PowerPC ISA ports each took about 1 month. Once again, about half of this time can be attributed to adding missing features to the framework.

We discovered a handful of errors in most ISA reference manuals and we even found a few bugs in every external assembler. This could be determined by three-way comparisons between our assemblers, the external assemblers and reference manuals.

Even though there is no absolute certainty regarding the validity of our instruction encodings and decodings, we stipulate that the number of bugs that our system would contribute to a complex compiler system should be minimal.

For now we have been focusing on correctness, functionality and completeness, and we have not yet had enough time to analyze and tune the performance of any part of the PMAS.

Not having emphasized performance in the design of the disassemblers, generators and testers, we find it sufficient that the disassemblers produce pages of listings quicker than humans can read even a single line and that each generator runs maximally for tens of seconds.

With regard to assembler performance, we paid attention to avoiding impediments that would be difficult to remedy later. Our first impressions suggest that even without tuning our assemblers are fast enough for use in static compilers and in optimizing dynamic compilers with intermediate representations. Performance is clearly not yet adequate when emitting code in a single pass JIT, though. To remove the most obvious performance bottleneck, we plan to replace currently deeply stacked output routines with more efficient operations, e.g., from `java.nio`.

As future work, we envision generating source code for disassemblers and providing a more elegant programmatic disassembler interface to identify and manipulate disassembled instructions abstractly.

The full source code of the PMAS is available under a BSD license at [2]. We recommend direct CVS download into an IDE. Project files for both NetBeans and Eclipse are included. Complementary prepackaged assembler jar files are planned, but not yet available at the time of this writing.

8. ACKNOWLEDGEMENTS

Adam Spitz created a draft version for SPARC and PowerPC of the presented system by porting the Klein Assembler System from Self to the Java language.

We would like to thank Mario Wolczko and Greg Wright for their helpful comments, and their insightful perusal of our first draft. We'd also like to thank Cristina Cifuentes for sharing her insights on the NJMC toolkit with us.

9. REFERENCES

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, 1995.

[2] Bernd Mathiske and Doug Simon. Project Maxwell Assembler System [online]. 2006. Available from: `http://maxwellassembler.dev.java.net/`.

[3] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, and H. Srinivasan. The Jalapeno Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[4] P. P. K. Chiu and S. T. K. Fu. A generative approach to universal cross assembler design. *SIGPLAN Notices*, 25(1):43–51, 1990.

[5] Eclipse.org. C/C++ Development Tools [online]. 2003. Available from: `http://eclipse.org/cdt`.

[6] D. Elsner, J. Fenlason, and et al. Using the gnu AS assembler [online]. 1999. Available from: `http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html`.

[7] M. F. Fernandez and N. Ramsey. Automatic checking of instruction specifications. In *ICSE*, pages 326–336, 1997.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[9] Free Software Foundation, Inc. GNU Lightning [online]. 1998. Available from: `http://www.gnu.org/software/lightning`.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Massachusetts, 2005.

[11] IBM Corporation. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, second edition, 1994.

[12] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*, 2006. Available from: `ftp://download.intel.com/design/Pentium4/manuals/25366619.pdf`.

[13] P. Johnson and M. Urman. The Yasm Core Library: Libyasm [online]. 2003. Available from: `http://www.tortall.net/projects/yasm/wiki/Libyasm`.

[14] ovm.org. OVM [online]. 2005. Available from: `http://www.cs.purdue.edu/homes/jv/soft/ovm/`.

[15] I. Piumarta. The virtual processor: Fast, architecture-neutral dynamic code generation. In *Virtual Machine Research and Technology Symposium*, pages 97–110. USENIX, 2004.

[16] N. Ramsey and M. F. Fernandez. The New Jersey Machine-Code Toolkit. In *USENIX Winter*, pages 289–302, 1995.

[17] N. Ramsey and M. F. Fernandez. Specifying Representations of Machine Instructions. *ACM Trans. Program. Lang. Syst.*, 19(3):492–524, 1997.

[18] Sun Microsystems. The Java HotSpot Virtual Machine, 2001. Technical White Paper.

[19] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pages 11–20. ACM, 2005.

[20] D. L. Weaver and T. Germond. *The SPARC Architecture Manual - Version 9*. Prentice-Hall PTR, 1994.

[21] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Sci. Comput. Program*, 57(3):339–356, 2005.

[22] J. D. Wick. *Automatic Generation of Assemblers*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975.

Tatoo: an innovative parser generator

Julien Cervelle
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
julien.cervelle@univ-mlv.fr

Rémi Forax
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
remi.forax@univ-mlv.fr

Gilles Roussel
Université de Marne la Vallée
Institut Gaspard-Monge
UMR CNRS 8049
77454 Marne-la-Vallée France
gilles.roussel@univ-mlv.fr

ABSTRACT

This paper presents Tatoo, a new parser generator. This tool, written in Java 1.5, produces lexer and bottom-up parsers. Its design has been driven by three main concerns: the ability to use the parser with the non-blocking IO API; the possibility to simply deal with several language versions; a clean separation between the lexer definition, the parser definition and the semantics. Moreover, Tatoo integrates several other interesting features such as lookahead-based rule selection, pluggable error recovery mechanism, multiple character sets optimized support.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Parser generator

1. INTRODUCTION

This paper presents Tatoo, a new parser generator. More precisely, given a set of regular expressions describing tokens, a formal specification of a grammar and several semantic hints, Tatoo, like many other existing tools [13, 10], can generate a lexer, a parser and several implementation glues allowing to run a complete analyzer that creates trees or computes simpler values. Thanks to a clean separation between specifications, the lexer and the parser may be used independently. Moreover, there implementations are not strongly linked to a particular semantic evaluation and may be reused, in different contexts, without modification.

Tatoo is written in Java 1.5 and heavily uses parameterized types. Currently, generated implementations are also Java 1.5 compliant but thanks to the simple extension mechanism of Tatoo, other back-ends may be provided for different target languages or implementations. This extension mechanism is already used to produce different lexer and parser implementations. For the lexing, depending on the character encoding, Tatoo provides three implementations: interval based, table based and *switch-case* based. For the parsing, depending on the grammar, SLR, LR(1) and LALR(1) implementations are available.

For the front-end, the Tatoo engine relies on reified lexer or parser specifications where regular expressions or productions are Java objects. Thus, everything may be implemented in Java. However, for usability, a simple XML front-end for these specifications is also provided.

For the Java back-end, a library is provided for runtime support. It contains generic classes for lexer and parser implementation, glue code between lexer and parser, a basic AST support and some helper classes that ease debugging. Generated Java code uses parameterized types and enumeration facilities to specialize this runtime support code. Moreover, all memory allocations are performed at creation time.

One main feature of the Java back-end resides in its ability to work in presence of non-blocking inputs such as network connections. Indeed, the Tatoo runtime supports push lexing and parsing. More precisely, contrarily to most existing tools, Tatoo lexers do not directly retrieve "characters" from the data input stream but are fed by an input observer which may work on non blocking inputs. The lexer retains its lexing state and resumes lexing when new data is provided by the observer. When a token is unambiguously recognized the lexer pushes it to the parser in order to perform the analysis. It is easy to provide a classical pull implementation based on this push implementation.

Another innovative feature of Tatoo is its support of language versions. This is provided by two distinct mechanisms. First, productions can be tagged with a version and the Tatoo engine produces a shared parser table tagged with these versions. Then, given a version provided at runtime, the parser selects dynamically the correct actions to perform. Second, the glue code linking the lexer and the parser supports dynamic activation of lexer rules according to the parser context. This feature allows, in particular, the activation of keywords according to the version.

Moreover, Tatoo integrates several other interesting features such as a pluggable error recovery mechanism and multiple character sets optimized support.

The rest of this paper is organized as follows. An overview of Tatoo development process is given in section 2. Then, in section 3, innovative features of Tatoo are detailed. Related works are presented in section 4, before conclusion.

2. TATOO OVERVIEW

2.1 General presentation

In this section a general overview of the typical construction of an analyzer using Tatoo is detailed. This process is divided into three steps:

- specification of the language to be parsed and of several semantic hints;
- automated implementation of parsing mechanism according to the chosen method and target language;
- implementation of the semantics.

2.2 Language specification

In order to simplify the bootstrapping of our system, we have chosen to describe the language using XML files, and a SAX-based XML parser. The specification of a language is provided by three files: one for the lexer (`.xlex`), one for the parser (`.xpars`) and one for the glue between them and a semantic part (`.xtls`).

For each file item (lexer rule, terminal, non-terminal, production, version, ...), a corresponding *id* is defined. At runtime, those items are referenced using this id.

In order to have the grammar unpolluted by code or semantic information, the grammar and the lexer specifications are clearly separated from the semantics, so that the same language can be used with different semantics.

2.2.1 Lexer

The lexing process consists in cutting the input text in lexical units called tokens. The specification file defines rules which are matched against the input. The lexer forwards the recognized tokens to a *listener* which, may or may not, send them to the parser.

2.2.1.1 Rules.

Each rule is described by one *main* regular expression and, optionally, a *following* regular expression. A rule is matched if the input matches the former expression and is followed by a sequence which matches the latter regular expression, if provided.

Regular expression can be expressed either using a Perl-like syntax [4], or using an XML syntax, the latter being mainly used for bootstrapping. For instance, an identifier is defined in XML by:

```
<rule-xml id="identifier">
  <main>
  <!-- definition of main regex -->
   <cat>
    <set>
     <interval from="a" to="z"/>
     <interval from="A" to="Z"/>
     <letter value="_"/>
    </set>
    <star>
     <set>
      <interval from="a" to="z"/>
      <interval from="A" to="Z"/>
      <interval from="0" to="9"/>
      <letter value="_"/>
     </set>
    </star>
   </cat>
  <!-- no follow expression -->
</rule-xml>
```

and in Perl-like by:

```
<rule id="identifier"
      pattern='[a-zA-Z_][a-zA-Z0-9_]*'/>
```

At runtime, when a particular rule is recognized, its *id* is forwarded to the listener using a back-end dependent type. The default Java back-end uses Java 1.5 enumeration types. The character sequence corresponding to the token is also provided, but it is not extracted from the input. It is the responsibility of the programmer to retrieve it, if needed.

As in other lexer generators, support for macros is also provided using `define-macro-xml` and `define-macro` XML tags.

2.2.1.2 Activator.

In order to speed up lexing process, a mechanism is provided to only select a subset of "useful" rules. The lexer is constructed using an object implementing the interface `RuleActivator` that selects, depending of the context, which rules are active.

One possible use of this mechanism is the implementation of Lex [12] start conditions, but as explained in section 3.2.1 it is also used to support language evolutions.

2.2.2 Parser

The parsing process consists in verifying that the flow of tokens produced by the lexer respects a particular organization described by a formal grammar.

The parser file specifies this grammar. It is described by the declaration of its productions, allowing an extended syntax of BNF using one-depth regular expressions (lists and optional elements). One or more starts (or axioms) for the grammar must be declared. If more that one axiom is declared for the grammar, the "real" axiom is precised to the parser at runtime.

For instance, Java method headers can be declared using the following production:

```
<production id="header">
  <lhs id="header"/>
  <rhs>
     <list id="modifiers" empty="true"
           element="methodModifier"/>
     <right id="returnType"/>
     <right id="identifier/>
     <right id="leftPar"/>
     <list id="parameters" empty="true"
           element="variableDeclaration"
           separator="comma"/>
     <right id="rightPar"/>
     <right id="throwsDeclaration"
            optional="true"/>
  </rhs>
</production>
```

`right` means a single or possibly no occurrence of a terminal or non-terminal ; `list` means a list of terminals and non-terminals.

To resolve common conflicts, as in Yacc [9], priorities can be associated with productions and terminals. For simplicity, priorities of productions are automatically deduced from priorities of their terminals, if applicable.

In the grammar file an error terminal is also declared for error recovery (see section 3.3.2) and versions may be specified for productions (see section 3.2.2). Moreover, the parser can provide, back to the lexer, the set of expected terminals at each step of parsing (see section 3.2.1).

2.2.3 Tools

A complete language analyzer uses lexer and parser processes to perform semantic computations according to a particular input stream. The tools file describes how the lexer and the parser work together and their interfaces with the semantics.

First, the association between rules and terminals is described. This piece of information is declared for two purposes. On the one hand, Tatoo generates automatically a lexer listener which forwards the corresponding terminal to the parser. On the other hand, it generates a lexer activator that selects only useful rules according to the expected set of terminals provided by the parser (see section 3.2.1). Rules that have to be always selected (like blank skipping rule) are those not associated with any terminal.

For instance the following lines associate rule `identifier` with terminal `identifier` and says that `space` are not pushed to the parser.

```
<rule id="identifier" terminal="identifier"/>
<rule id="space" spawn="false"/>
```

Second, like in Yacc, the tools file declares the type (primitive or not) of the attribute associated with each terminal and each non-terminal.

For instance, following lines associate the type `String` with the terminal `identifier` and the type `int` with non-terminals `ref` and `expr`.

```
<terminal id="identifier" type="String"/>
<non-terminal id="ref" type="int"/>
<non-terminal id="expr" type="int"/>
```

This piece of information is used to generate automatically a lexer listener. The listener receives, at creation time, an object provided by the developer that computes the terminal attribute value from the character sequence matched by the rule. This object specify a particular semantics for terminal attributes implementing the interface `TerminalAttribute-Evaluator` that is generated by Tatoo. Hence, the semantics can be modified without regenerating the lexer classes.

For our running example, tools generator produces the interface below:

```
interface TerminalAttributeEvaluator {
  void space(CharSequence seq);
  String identifier(CharSequence seq);
}
```

Types of non-terminals and terminals are also used to build a parser listener which performs semantic actions. Like for the lexer listener, the developer has to provide, at creation time, an implementation of an interface called `Grammar-Evaluator` generated by Tatoo. For all productions this implementation constructs the attribute of the left hand side non-terminal from attributes of the right hand side elements. An implementation for the attribute stack and for shift handling are provided to simplify this implementation.

Here again, since the semantic computations conform to a predefined interface, changing the behavior of the parser does not require to reprocess lexer and parser files. Moreover, different semantics may be attached to the very same generated lexer and parser.

For example, using previous tools specification and a grammar like the following:

```
<start id="expr"/>
<production id="p1">
 <lhs id="expr"/>
 <rhs>
  <right id="expr"/>
  <right id="plus"/>
  <right id="expr"/>
 </rhs>
</production>
<production id="p2">
 <lhs id="expr"/>
 <rhs><right id="ref"/></rhs>
</production>
<production id="p3">
 <lhs id="ref"/>
 <rhs>
  <right id="excl_mark"/>
  <right id="identifier"/>
 </rhs>
</production>
```

the interface generated by tools generator is:

```
interface GrammarEvaluator {
  int p1(int expr, int expr2);
  int p2(int ref);
  int p3(String indentifier);
  void acceptExpr(int expr);
}
```

In this generated interface, there is one method for each production. Each one is strongly typed according to types associated with terminals and non-terminals in the tools file. Moreover, one `accept()` method is generated for each axiom non-terminal. Here, the only axiom specified for the grammar is `expr`.

Finally, tools generator offers the possibility, if needed, to produce a special semantic evaluator that performs the construction of an abstract syntax tree. In this tree, each node kind has a corresponding Java type. It allows efficient traversal of the tree using the visitor design pattern (see section 3.3.3).

2.3 Generation

Tatoo provides pluggable generators in order to build code for any target language. The Tatoo engine builds all the tables and maps needed to write the lexer, parser and tools code

- the transition tables of each finite automaton;
- the action table for the parser (according to state and lookahead);
- maps between various objects (versions and productions, rules and terminals...)

In order to generate class and interface files, these objects are made available in the generator, for back-end extensions, using a bus mechanism. The default Java back-end uses

Velocity [2], an Apache package that follows a model-view approach to generate files. More precisely, the Velocity language allows to specify template files used by the Velocity engine that directly interacts with Java objects provided by the Tatoo generator to produce output files.

2.4 Runtime

2.4.1 *Memory allocation*

One main concern about the runtime was to prevent the parser and the lexer from creating any new object. Thus, all memory required by the runtime is allocated when the lexer and the parser are created. No more memory is allocated during execution, except potential parser stack extensions. This feature is important in order to allow to embed Tatoo parsers in long-living applications such as web servers.

2.4.2 *Lexer*

The lexer is built on top of a buffer used to store available characters from the input stream to be processed. When new characters are made available in this buffer by an external mechanism (such as a file reader or a network observer), the `step()` method of the lexer may be called. Then, all available characters in the buffer will be examined. For each character, once all rules have been applied, the lexer verify if a lexical unit has been completely recognized. When multiple rules match, the lexer uses the following priority: first the longest match, and then the first one declared (this behavior conforms to Lex [12]). Each time a “winning” rule is unambiguously found, the lexer listener’s only method, `ruleVerified()`, is called with this rule and an object implementing the interface `TokenBuffer` as arguments. The interface `TokenBuffer` provided a method `view()` to access (as a `CharSequence`) the recognized lexical unit and a method `discard()` to indicate that recognized characters may be pulled off the lexer input buffer (they are not discarded automatically).

2.4.3 *Parser*

Within the lexer listener generated by the tools generator, terminals are passed, when required, to the parser using its method `step()`. This method performs all the actions it can using this new input terminal (possibly some reduces and one shift). Like for the lexer, a listener is attached to the parser at creation time, and for each action, the listener is called in order to perform the semantic part. The parser listener has three methods, one for each kind of action: `shift()` which means a terminal is read; `reduce()` which means a production is applied and its left hand side replaces its right hand side; and `accept()` when the input is accepted for a specific axiom. To trigger acceptance, one has to call the `close()` method. It happens when the lexer reaches an “end-of-file” condition.

All these gears are described in figure 1.

Together with these mechanisms, error recovery (see section 3.3.2), selection of valid rules according to expected terminals and support for language versions (see section 3.2.2) are also available.

3. INNOVATIVE CONCERNS

3.1 Non blocking parsing

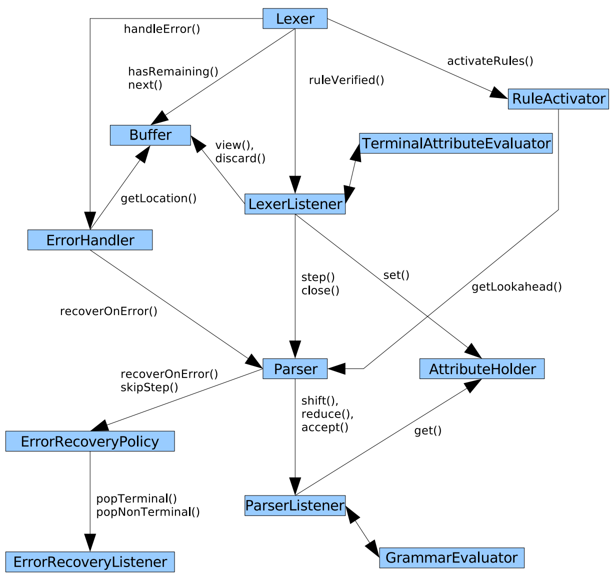


Figure 1: Lexer, parser and tools interactions

The primary goal of the development of Tatoo was to provide a parser generator compatible with non-blocking IO. Indeed, popular existing parser generators only produce lexers and parsers working with blocking API. These extract data from the stream and wait until data is available. This behavior is known as pull parser (see figure 2) and calls to the parser are blocking. This behavior is acceptable for parsers since they usually work on files where blocking periods are small, but this is not compatible with network streams where waiting periods may be long. In order to support an efficient parsing of network input streams Tatoo supports push lexers and parsers.

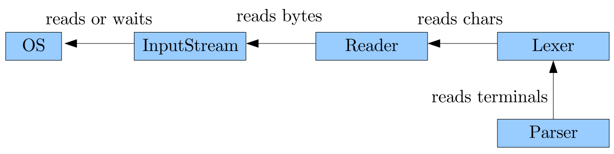


Figure 2: Pull parsers and lexers

3.1.1 *Push behavior*

In the push behavior (see figure 3), the lexer and the parser processes maintain a state. These processes are started explicitly when data is available and stopped when no more data remains in the input buffer which does not mean that the end of the stream is reached. Indeed, data may arrive later and thus the end of the stream has to be notified explicitly. Thus, the basic API of our push lexers and parsers contains two methods: `step()` and `close()`. The method `step()` is used to push characters or terminals and the method `close()` indicates the end of the stream. A method `run()` is provided for convenience to simulate pull behavior. It just performs a read/step loop until the end of the file is reached and then it closes the lexer that closes the parser.

3.1.2 *Common abstract buffer API*

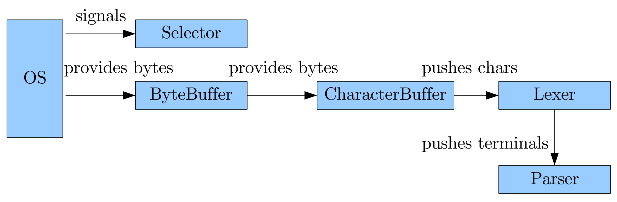


Figure 3: Push parsers and lexers

Lexers generated by Tatoo can run on several kinds of buffer. Indeed, the lexer only requires four methods encapsulated in the interface `ChararacterBuffer` to be implemented. These four methods are: `next()`, `hasRemaining()`, `unwind()` and `previousWasNewLine()`. The first method returns the next character in the buffer; the second tells whether characters remain in the buffer; the third is used by the lexer to indicate that a token is recognized and that the lexing has to resume; the last one is used to select rules that require to occur at the beginning of a line.

Tatoo runtime provides several wrappers implementing this interface for Java NIO buffers, readers and input streams.

3.2 Language evolution

In order to follow language evolutions such as those of Java, Tatoo introduces two special mechanisms. The first one is called lookahead activator and it simplifies keyword additions. The second, called grammar versioning, permits to specify, in a single grammar, different language constructions tagged with their version.

3.2.1 Lookahead activator

Like several other lexer generators such as Lex [12], the lexer specification is composed of multiple rules and of an activator to activate or deactivate these rules during analysis. The activator is usually implemented by the developer to implement start conditions in the lexer. In Tatoo, by default, the tools generator implements a special activator based on the terminals expected by the parser, known as lookahead. More precisely, for each state of the push-down automaton, the parser only provides the terminals that do not cause syntax errors. These terminals are associated with particular lexer rules that the generated activator retains, deactivating other rules.

This feature allows first to speed up the lexing process since usually only few terminals/rules are selected in each parser state. Second, and more importantly, it allows to minimize conflicts in existing source codes when a new keyword is introduced in the language. Indeed, if the new keyword only appears in a particular production of the grammar, the lexing rules matching this keyword is not selected in other productions, and the keyword can be recognized as an identifier. For instance, the `enum` new keyword of Java 1.5 potentially conflicts with identifiers of previous versions. However, since `enum` keyword is never acceptable where an `enum` identifier is acceptable, a Java parser that uses the lookahead activator could suppress such conflicts.

This approach suppresses parser process errors. Indeed, the lexer never generates an unexpected terminal. Thus, it disables the classical grammar-based error recovery mechanisms. This is why the Tatoo tools generator translates lexing exceptions into parser errors, pushing a special error terminal to the parser.

3.2.2 Grammar version

The parser file allows to associate a particular version with each production and specify a single inheritance relation between these versions. The version hierarchy tree is specified by declaring for each version an optional implied version. The implication relation between versions is transitive. Each production can only declare one version, but if this version is implied by other versions, it is used by all these versions.

For instance, the following portion of parser file specifies two different versions `v1` and `v2` linked by an implication relation and two productions `p1` and `p2` with their respective versions. Since, `v1` is implied by `v2`, production `p1` is active for version `v2`.

```
<version id="v1"/>
<version id="v2" implies="v1"/>

<production id="p1" version="v1">
 <lhs id="A"/><rhs><right id="a"/></rhs>
</production>

<production id="p2" version="v2">
 <lhs id="A"/><rhs><right id="b"/></rhs>
</production>
```

Given a versioned parser specification, first, the parser generator computes the push-down automaton according to the grammar, using a bottom-up algorithm (SLR, LR(1) or LALR(1)) taking into account neither versions nor priorities. Afterwards, for each version, the Tatoo engine selects transitions compatible with this version. More precisely, a reduce transition is compatible with a version if the target production is active for the selected version. A shift transition is compatible with a version if one of the kernel items (productions) of its target state is active. Then, for each version, the classical conflict resolver based on priority and associativity determines the unique action to perform. Lastly, in order to reduce action lookup and the size of the table, if all versions lead to a same action, version information are not saved in the table. Otherwise an action is saved for each version using a composite action. The "real" version, provided at runtime, allows to select the correct action to perform. Figure 4 illustrates all these steps.

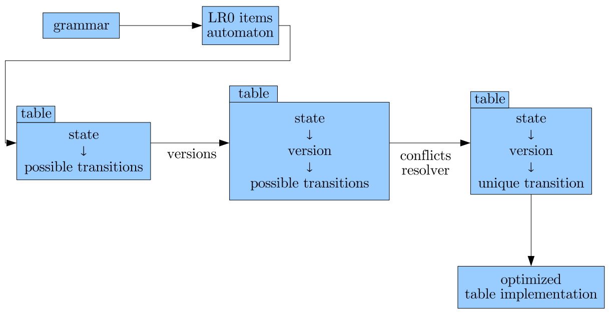

grammar
LR0 items automaton
table
state ↓ possible transitions
versions
table
state ↓ version ↓ possible transitions
conflicts resolver
table
state ↓ version ↓ unique transition
optimized table implementation

Figure 4: Parser table construction with versions

3.3 Other features

Even if non-blocking API and grammar evolution supports were primary goals for the development of Tatoo, generated analyzers also support several other interesting features.

3.3.1 *Charset selection*

In order to support multiple charsets, Tatoo can rely on the Java `java.util.Reader` mechanism. However, it is also possible to encode, at generation, the lexer tables in the target charset of the input stream. This mechanism avoids unnecessary decoding of characters at runtime. The drawbacks of this approach are that lexer implementations is linked with a particular charset and that interval specification in regular expressions, such as `[a-d]`, depends of the character encoding. However, these intervals should only concern letters or digits whose intervals do not depend on the charset.

Another optimization that depends on the charset is the lexer implementation of rule automata. The classical implementation of transitions in these automata uses character intervals since for unicode charset they cannot be implemented directly using a character array. Two other implementations are provided. The first one uses arrays if the charset is small enough. The second one uses a *switch-case* implementation and is valuable if most transitions can be implemented by the default case, which is often the case for programming languages tokens.

3.3.2 *Pluggable error recovery*

As explained in section 3.2, only lexer errors may occur during analysis, but these errors are forwarded to the parser in order to trigger an error recovery mechanism. This mechanism is plugged into the parser, at creation time, by the developer. The default algorithm provided to the parser does not perform any error recovery.

However, it is quite simple (even if less user-friendly than for LL(k) parser) to develop an error recovery mechanism, specific to a particular context, implementing the abstract class `ErrorRecoveryPolicy`. Two main methods have to be implemented: `recoverOnError()` and `skipStep()`. When an error occurs, the method `recoverOnError()` is called with the parser state in order to recover from the error. Then, either the error is recoverable and parsing may continue, or parsing is aborted. Moreover, the method `skipStep()` is called each time a terminal is pushed to the parser. It can be used by the error recovery mechanism to indicate that this terminal must be ignored; for instance to skip all terminals until a semicolon is found.

We provide the implementation of two classical error recovery algorithms. The first one is described in [1] and used by Yacc [9] and the second one is similar to the ANTLR [13] error recovery algorithm.

Because these two algorithms can pop a state out of the stack, the implementation uses a specific listener, `ErrorRecoveryListener`, in order to signal to the semantic part that the error recovery mechanism has popped a state.

3.3.3 *Generic AST construction*

Abstract tree construction is an optional feature of the tools generator that does not require extra specification. The AST generator uses the tools file and it produces a specific semantic evaluator linked with the parser.

Like in SableCC [5], types of nodes are directly deduced from the grammar. Furthermore, these types accept visitors [6] for semantic evaluations.

However, Tatoo generated types are particular since they allow two views of the same node. For the first one, Tatoo creates a specific set of classes and interfaces directly deduced from the grammar specification. This view is comparable with the tree implementation produced by the JAXB [14] approach for XML document. There is one interface for each non-terminal and a concrete class for each production. The right hand side of the production is implemented using attributes. Their types are as precise as possible.

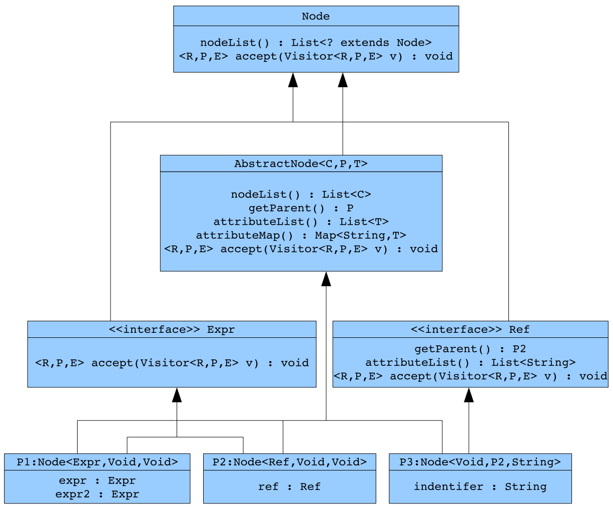

Node
nodeList() : List<? extends Node>
<R,P,E> accept(Visitor<R,P,E> v) : void
AbstractNode<C,P,T>
nodeList() : List<C>
getParent() : P
attributeList() : List<T>
attributeMap() : Map<String,T>
<R,P,E> accept(Visitor<R,P,E> v) : void
<<interface>> Expr
<R,P,E> accept(Visitor<R,P,E> v) : void
<<interface>> Ref
getParent() : P2
attributeList() : List<String>
<R,P,E> accept(Visitor<R,P,E> v) : void
P1:Node<Expr,Void,Void>
expr : Expr
expr2 : Expr
P2:Node<Ref,Void,Void>
ref : Ref
P3:Node<Void,P2,String>
indentifer : String

Figure 5: Example of AST type hierarchy

The second view allows a generic access to the tree comparable with the XML DOM [11] approach. Tree nodes are viewed through the common interface `Node`. Its method `nodeList()` returns the read-only list of its children and its method `getParent()` returns its parent. The interface `Node` is implemented by all specific classes generated by Tatoo and a generic list of children is lazily generated from their attribute list, on demand.

For instance, given the following grammar in BNF notation, the type hierarchy represented in figure 5 is generated by Tatoo tools.

```
p1: expr ::= expr '+' expr
p2: expr ::= identifier
p3: identifier ::= '!' 'identifier'
```

Generated trees accept two kinds of visitor: a generic one and a specific one that respectively correspond to the different views of the tree. Here is a simplified version of the generic visitor `NodeVisitor`:

```
class NodeVisitor<R,P,D,E extends Throwable> {
 R visit(Node node,P param) throws E {
  throw new RuntimeException();
 }
}
```

This visitor is parameterized by the signature of the method `visit()` and only permits a simple traversal of the tree.

A strongly typed visitor of the generated type hierarchy is also produced. It inherits from `NodeVisitor` and, by default, its methods `visit()` delegate their implementation to a less precise method according to the type hierarchy.

For instance, if we consider the previous grammar and the type hierarchy of figure 5, the generator produces a visitor that contains the following methods:

```
R visit(P1 p1, P param) throws E {
 return visit((Expr)p1, param);
}
R visit(Expr e, P param) throws E {
 return visit((Node) e, param);
}
```

This delegation model allows developers to implement general behavior for some types and specific ones for others. More precisely, they do not have to implement all methods `visit()` if one of the super-type visitor captures all behaviors of its subtypes. For instance, if the behavior of the visit is the same for nodes of types `P1` and `P2`, the developer only has to implement the method `visit(Expr e,...)`.

3.3.4 Runtime lexer and parser

By default, the lexer and the parser are generated offline by the Tatoo generators using XML specifications. However, it is possible to construct them entirely at runtime specifying everything in Java. Indeed, XML specifications are only available for convenience. Internally, rules or productions are reified into Java objects that the developer may construct directly.

For instance, the following portion of code creates a rule with id `value` recognizing numbers using a rule factory. A similar factory is also available for grammar dynamic specification.

```
RuleFactory rf = new RuleFactoryImpl();
Encoding charset = LexerType.unicode.getEncoding();
Map<String,Regex> map =
  Collections.<String,Regex>emptyMap();
PatternRuleCompilerImpl ruleCompiler=
  new PatternRuleCompilerImpl(map,charset);
ruleCompiler.createRule(rf,"value","[0-9]+");
```

Then, given the description of rules, it is possible to directly obtain a lexer without intermediate code generation. However, lexer creation induces several costly computations at runtime, such as automata minimization.

The following portion of code constructs a lexer given a rule factory `rf`, a character buffer `b` and a lexer listener `l`.

```
SimpleLexer lexer =
   RuntimeLexerFactory.createRuntimeLexer(rf,b,l);
```

The lexer constructed this way is then almost as efficient as offline-constructed lexers.

4. RELATED WORKS

In this section, the overview of existing compiler generators is restricted to those written in Java and producing Java implementations. Of course, many other tools exist for other languages.

4.1 JavaCC

Developed by Sun Microsystems, Java Compiler Compiler [10] is a parser generator written in Java generating top-down parsers.

JavaCC by default only works for LL(1) grammars and let the user annotate the grammar with a specific k lookahead to resolve ambiguities.

The semantic part of the parser is directly integrated in the grammar specification. The actions are specified in Java and some keywords allow to obtain parsing information. The fact that grammar and semantic part are mixed in the same file is error prone with large grammar and does not permit to reuse the grammar without its semantics.

4.2 ANTLR

ANother Tool for Language Recognition [13] written by Terence Parr, is a parser generator generating top-down compilers from grammatical specifications with back-ends in Java, C#, C++, or Python.

Like JavaCC, ANTLR is a LL(k) parser generator. The grammar is augmented with actions specifying the semantics of the compiler or translator. ANTLR provides a domain specific language for tree construction, tree walking and tree transformation.

4.3 JFLex/Cup

JFlex [7] and Cup [8] are ports of Lex and Yacc in Java. JFlex is a lexer generator in Java that reuses the same regex format than Lex on the unicode charset. Cup generates bottom-up compilers from LALR(1) grammar using a syntax similar to Yacc. Like JavaCC, it mixes the grammar specification and the semantic actions specified in Java.

4.4 SableCC

Sable Compiler Compiler [5] was initially written by Etienne Gagnon during its master thesis. It generates a bottom-up compiler from a LALR(1) grammar. However, it does not allow to resolve conflicts with associativity and priorities.

SableCC does not permit to specify semantics in the grammar but it generates an AST letting the developer rely on visitors to express the semantics. Thus it provides a clean separation between grammar specification and semantics. However, even if the generated AST is tweakable there is no way to generate a specifc tree without creating another AST.

4.5 Beaver

Beaver [3] is an LALR(1) parser generator that claims to generate fast compilers.

Like Tatoo, beaver generates only a parser table and relies on a runtime parser. Furthermore the table contains actions to perform using late-binding call that frequently outperform the traditional switch implementation.

5. CONCLUSION

In this paper we have presented Tatoo a new compiler generator written in Java which main innovative features are:

- push lexing and parsing in order to support non blocking IO;
- support for grammar evolutions.

Tatoo has been used this year for compilation courses. During these courses, students have implemented a compiler for a simple language that produces Java bytecode. We did not encounter special difficulties induces by Tatoo's specific architecture compared with SableCC used previous years. However, we still have to implement a parser for a real language such as Java or C and to perform benchmarks to prove usefulness of Tatoo features.

In order to enhance Tatoo usability, we plan to support complete EBNF specification. Second, we also want to add automatic documentation comparable to Javadoc in input specifications. Finally, other language back-ends, such as C++ and C#, will be added.

The latest stable version of Tatoo is freely available on the Web and can be downloaded from `http://tatoo.univ-mlv.fr/`.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Apache Software Foundation. Velocity 1.4 User Guide. `http://jakarta.apache.org/velocity/`, Jan. 2006.

[3] A. Demenchuk. Beaver - a LALR parser generator. `http://beaver.sourceforge.net/index.html`, 2006.

[4] J. E. F. Friedl. *Mastering Regular Expressions.* O'Reilly, 2^{nd} edition, July 2002.

[5] E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *Technology of Object-Oriented Languages and Systems*, pages 140–154. IEEE Computer Society, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[7] K. Gerwin. *JFlex User's Manual*, July 2005.

[8] S. E. Hudson. *CUP User's Manual.* Usability Center, Georgia Institute of Technology, July 1999.

[9] S. C. Johnson. Yacc: Yet Another Compiler Compiler, 1979.

[10] V. Kodaganallur. Incorporating language processing into java applications: A JavaCC tutorial. *IEEE Software*, 21(4):70–77, Aug. 2004.

[11] A. Le Hors and P. Le Hégaret. Document object model level 3 core. `http://www.w3.org/DOM/DOMTR`, Apr. 2004.

[12] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical Report 39, Bell Laboratories, July 1975.

[13] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[14] Sun Microsystems Inc. Java(TM) architecture for XML binding specification. `http://java.sun.com/xml/downloads/jaxb.html`, Jan. 2003.

Session B
Program and Performance Analysis

Cost and Benefit of Rigorous Decoupling with Context-Specific Interfaces

Florian Forster
University of Hagen
Universitätsstraße
D-58097 Hagen
+49 (0) 2331 987-4290
florian.forster@fernuni-hagen.de

ABSTRACT

In Java programs, classes are coupled to each other through the use of typed references. In order to minimize coupling without changing the executed code, interfaces can be introduced for every declaration element such that each interface contains only those members that are actually needed from the objects referenced by that element. While these interfaces can be automatically computed using type inference, concerns have been raised that rigorous application of this principle would increase the number of types in a program to levels beyond manageability. It should be clear that decoupling is required only in selected places and no one would seriously introduce a minimal interface for every declaration element in a program. Nevertheless we have investigated the actual cost of so doing (counted as the number of new types required) by applying rigorous decoupling to a number of open source Java projects, and contrasted it with the benefit, measured in terms of reduced overall coupling. Our results suggest that (a) fewer new interfaces are needed than one might believe and (b) that a small number of new interfaces accounts for a large number of declaration elements. Particularly the latter means that automated derivation of decoupling interfaces may at times be useful, if the number of new interfaces is limited a priori to the popular ones.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: *Modules and interfaces*

General Terms
Design

Keywords
The Java Language::Java-specific metrics
The Java Language::Optimization
Software Engineering with Java::Tools for Java Programming

1. INTRODUCTION

Interface-based programming as described in [1] is accepted as a useful object-oriented programming technique. According to it, references in a program should be declared with interfaces and not classes as their types. Two main benefits can be expected from this: First, flexibility is increased as classes implementing an interface can be exchanged without notification of the client using the classes' services via the interface. Second, the access to the class is restricted to the methods declared in the interface typing a variable which holds a reference to an instance of the class. While these interfaces can be automatically computed using type inference, concerns have been raised that rigorous application of this principle would increase the number of types in a program to levels beyond manageability

Nevertheless we have investigated the actual cost of so doing (counted as the number of new types required) by applying rigorous decoupling to a number of open source Java projects, and contrasted it with the benefit, measured in terms of reduced overall coupling.

The rest of the paper is organized as follows. In section 2 we will briefly introduce the refactoring and the metrics used in this paper, which already have been described in [10]. In section 3.1 we introduce our test suite, which consists of six open source projects. Afterward, we outline the initial situation before the refactoring using various metrics in section 3.2. The same metrics are applied on the projects after the refactoring in section 3.3. In section 3.4 we discuss the costs of the refactoring in terms of new types required in contrast to the reduced overall coupling. In section 3.5 we present additional insights gained during the investigation of our test suite. Section 4 recapitulates our results; section 5 concludes and provides pointers to future work.

2. THE REFACTORING

2.1 Measuring Coupling

Typing rules in Java enforce the type of a declaration element to offer at least the set of methods invoked on that declaration element. Coupling between classes increases to levels beyond what is necessary, when a declaration element is declared with a type offering more methods than actually needed by this declaration element. In the following we use the goal question metric approach to derive a suitable metric measuring unnecessary coupling.

The methods publicly available for a type $\boldsymbol{T}$ are a subset of all the methods declared in $\boldsymbol{T}$ or one of its supertypes[1]. Let $\mu(\boldsymbol{T})$ be the set of methods declared in a type T. Then we declare

[1] In Java we do not count the methods inherited from *Object*.

Table 1: Use of interfaces and decoupling quality in six open source projects before the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
Types	49 (100%)	73 (100%)	103 (100%)	118 (100%)	346 (100%)	395 (100%)
Classes	32 (65%)	48 (66%)	77 (75%)	95 (80%)	282 (82%)	301 (76%)
Interfaces	17 (35%)	25 (34%)	26 (25%)	23 (20%)	64 (18%)	94 (24%)
Ratio	1:0,53	1:0,52	1:0,34	1:0,24	1:0,23	1:0,31
Declaration Elements	190	166	278	468	1361	1801
Class typed	190	71	244	450	1109	395
Interface typed	0	95	34	18	252	1406
Ratio	-	1:1,338	1:0,139	1:0,040	1:0,227	1:3,560
ACD						
Average	0,365	0,216	0,447	0,567	0,416	0,142
Highest	0,813	0,997	1	0,998	1	1
Lowest	0	0	0	0	0	0
BCD						
Average	0,341	0,211	0,336	0,484	0,351	0,100
Highest	0,813	0,992	0,961	0,998	0,996	0,998
Lowest	0	0	0	0	0	0
ACD-BCD						
Average	0,024	0,005	0,110	0,084	0,065	0,041
Highest	0,222	0,326	1	0,795	1	1
Lowest	0	0	0	0	0	0

$$\pi(T) := \{m \in \mu(T) \mid m \textit{ is a public nonstatic method}\}$$

as the set of methods offered by a type ***T*** to all declaration element typed with ***T***.

In strongly typed languages like Java, each declaration element has a declared type. Besides its declared type, each declaration element also has an inferred type. Here, we define the inferred type of a declaration element ***d*** as the type ***I*** whose set of publicly available methods π(***I***) is the smallest set containing all methods directly accessed on ***d***, united with the sets of methods accessed on all declaration elements ***d*** possibly gets assigned to. We define π(***I***) as a function of ***d***, ι(***d***), which can be computed by analyzing the source code using *Static Class Hierarchy Analysis* as described in [9]. We call this inferred type ***I*** a *context-specific interface* or *minimal interface* for ***d*** and ι(***d***) the *access set* of ***d***.

In order to reduce unnecessary coupling between classes we have to redeclare every declaration element ***d*** with the type ***I*** so that π(***I***)=ι(***d***), i.e. the new declared type of ***d*** offers only those methods which are actually invoked on ***d*** or one of the declaration elements ***d*** possibly gets assigned to. According to the goal question metric approach described in [14] we can define a metric for coupling. Our question is how much unnecessary coupling is introduced to a type ***T*** by means of a declaration element ***d***, and our goal is to reduce this unnecessary coupling. We therefore define the coupling induced by a declaration element ***d*** as the quotient of the number of methods needed from ***d***, and the number of methods provided by its declared type ***T***:

$$\frac{|\iota(d)|}{|\pi(T)|}$$

A value of 1 indicates that ***d*** is declared with the least specific (most general or *minimal* in the sense that it has as few members as possible) type ***T***, i.e. the type offering only those methods actually needed, whereas one of 0 implies that none of **T**'s methods are used.

Obviously 1 – the quotient is a measure of the possible reduction of the unnecessary coupling established by the declaration element ***d*****,** if ***d*** is redeclared with a type ***I*** so that π(***I***)=ι(***d***). We have called this metric the *Actual Context Distance* (ACD) [10] for a declaration element ***d*** with type ***T*** and write

$$ACD(d,T) = \frac{|\pi(T)| - |\iota(d)|}{|\pi(T)|}$$

The *Best Context Distance* (BCD) for a declaration element ***d*** is then the lowest value of ACD achievable by redeclaring ***d*** with a type ***T*** which is already available in the project. It is a measure for the maximum decoupling that can be achieved by using only existing types.

We define ACD and BCD for a type ***T*** as the average ACD and BCD of all declaration elements typed with ***T***. The ACD and BCD for a project is the weighted average, the weight of a type ***T*** being the number of declaration elements typed with ***T***, of the ACD and BCD for each type of the project.

2.2 Refactoring for interface-based programming

In order to improve decoupling to the maximum, we can use the refactoring described in [10]. It reduces *ACD(**d**,**T**)* to zero for every declaration element ***d*** in a project. This is done by either

Table 2: Use of interfaces and decoupling quality in six open source projects after the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
Types	92 (100%)	110 (100%)	190 (100%)	260 (100%)	599 (100%)	709 (100%)
Classes	32 (35%)	48 (43%)	77 (41%)	95 (37%)	282 (47%)	301 (42%)
Interfaces	60 (65%)	63 (57%)	113 (59%)	165 (63%)	317 (53%)	418 (58%)
Ratio	1:1,88	1:1,31	1:1,47	1:1,74	1:1,13	1:1,39
Declaration Elements	177	124	267	447	1241	1655
Class typed	103	34	81	121	404	81
Interface typed	74	90	186	326	837	1574
Ratio	1:0,71	1:2,64	1:2,30	1:2,69	1:2,07	1:19,43
Declaration Elements per type	1,92	1,12	1,41	1,72	2,07	2,33

a) using an existing, better suited interface ***I*** for ***d*** with $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$, or by

b) introducing a new interface ***I*** for ***d*** tailored to fulfill $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$

for the redeclaration of ***d*** so that *ACD(**d**,**I**)*=0.

For the purpose of redeclaring ***d*** with its least specific type ***I***, no matter if this type already exists or is newly introduced, the type inference algorithm we use has to consider three cases.

First if the declared type ***T*** of a declaration element ***d*** is already minimal, i.e. $\pi(\boldsymbol{T})=\iota(\boldsymbol{d})$, no changes to the program are necessary.

If however $|\pi(\boldsymbol{T})|>|\iota(\boldsymbol{d})|$, i.e. ***d***'s declared type is not minimal, and ***d*** terminates an assignment chain, i.e. ***d*** is not assigned to any other declaration element, all we have to do is declare ***d***'s minimal type ***I***, i.e. $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$, as a supertype of ***T*** and redeclare ***d*** with ***I***.

Finally ***d***'s declared type might neither be minimal nor terminate an assignment chain. To illustrate this case we use a primitive scenario with two declaration elements ***A a*** and ***B b*** and the assignment ***b=a***.

Assuming that the program at hand is type correct and using the rules for typing in Java we know that the type ***B*** is the same type as **A** or a supertype of ***A***. Computing $\iota(\boldsymbol{a})$ gives us the new type ***I*** for the declaration element ***a***. Unfortunately redeclaring ***a*** with ***I*** results in a program which is not type correct, as the assignment of ***a*** to ***b***, i.e. ***b=a*** for ***I a*** and ***B b***, is undefined in Java, if ***I*** is not a subtype of ***B*** .

To solve this issue we can simply declare ***I*** as a subtype of ***B***. This makes the program type correct as ***A*** is a subtype of or equal to ***B*** and ***I*** is a subtype of or equal to ***A***. However introducing this relationship might renders ***I*** not to be a minimal type for ***a*** as it might add unwanted methods to ***I*** coming from ***B***, i.e. $\pi(\boldsymbol{B}) - \pi(\boldsymbol{I})$ is not empty.

As the introduction of this relationship might result in unwanted methods added to ***I***, we redeclare the declaration element ***b*** with some type ***J***, so that ***J*** is a supertype of ***I***, to make the program type correct again.

In case ***J*** is a real supertype of ***I***, i.e. $\pi(\boldsymbol{J}) \subset \pi(\boldsymbol{I})$, we have to make sure that ***J*** is declared as a supertype of ***I***. Furthermore in order to keep other assignments to ***b*** correct we have to make sure that ***J*** is declared as a supertype of ***B***.

2.3 Limitations of the Refactoring for Java

The implementation of the refactoring described in the last section has a few limitations due to Java's type system. If fields of a class are directly accessed using a declaration element, i.e. without using accessor methods, this declaration element can not be redeclared with an interface. Even though one could define an abstract class for the purpose of redeclaration we do not do so, as multiple inheritance is not possible in Java. Therefore this workaround would work only in a limit number of cases. Furthermore redeclaration of a declaration element with an interface is also not possible if nonpublic methods are accessed on the declaration element. Thus we excluded these declaration elements from our investigation as they can not be redeclared, i.e. unnecessary coupling does not exist.

The second limitation are empty interfaces. Declaration elements with empty access sets might be redeclared with empty interfaces as they are an ideal type. However empty interfaces, so called *tagging interfaces* or *marker interfaces*, are used in *instanceof b*oolean expressions in Java. Typing a declaration element with an empty interface might therefore lead to circumstances, in which the boolean expression evaluates to true after the redeclaration. To avoid these cases we rather redeclare declaration elements with empty access sets with the root of the type hierarchy, i.e. *Object* in Java. As only the number of declaration elements typed with types defined within the project is considered for the metrics, the declaration elements retyped with *Object* disappear. This is justified by the fact that every declaration element which is typed with *Object* has no influence on the coupling as a "coupling" with *Object* always exists due to the nature of Java, i.e. every type is subtype of *Object*.

Furthermore interfaces already existing in a project might become superfluous after the refactoring, i.e. no declaration elements are typed with these interfaces. However, even though we could, we do not delete these interfaces!

In the remainder of this paper we will use InferType on a number of open source projects, so that we can evaluate both the costs, in terms of newly introduced types, and the benefits, in terms of improved decoupling, of rigorous decoupling.

Table 3: Comparison of the situation before and after the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
ΔTypes	+43 / +88%	+37 / +50%	+87 / +85%	+142 / +105%	+253 / +73%	+314 / +80%
ΔInterfaces	+43 / --	+37 / +37%	+87 / +335%	+142 / +617%	+253 / +395%	+314 / +343%
ΔDeclaration Elements	-13 / -7%	-42 / -25%	-11 / -4%	-21 / -5%	-120 / -8%	-146 / -8%
ΔClass typed	-87 / -46 %	-37 / -52%	-163 / -66%	-329 / -73%	-705 / -64%	-314 / -80%
ΔInterface typed	+74 / +∞ %	-5 / -5%	+ 152 / +447%	+308 / +1711%	+585 / +232%	+168 / +12%
ΔACD average	-0,365	-0,216	-0,447	-0,567	-0,416	-0,142
ACD average per new type	-0,0084	-0,0057	-0,0051	-0,0040	-0,0016	-0,0004

2.4 Implementation of the Refactoring

The refactoring described in [10] was implemented and called *InferType*[2]. The algorithm used for applying the refactoring to a complete project is outlined below:

```
changes=false
do
  foreach type in project
    DEs:=getDeclarationElements(type)
    foreach DE in DEs
      refactor(DE)
      if(hasChanged(project))
        changes=true
      endif
    endfor
  endfor
while(changes)
```

For each type in the project we iterate over all the declaration elements declared with this type. We then apply the refactoring described in section 2.2 to each of these declaration elements. If there was a change, i.e. a new type was introduced to the project during the refactoring of a declaration element; we repeat the process until no more changes happen. After using this algorithm on a project every declaration element in this project is typed with a minimal type, i.e. *ACD(**d**,**T**)* is always zero.

3. ANALYSING RIGOROUS DECOUPLING WITH CONTEXT-SPECIFIC INTERFACES

3.1 Introducing the Test Suite

To evaluate the costs and benefits of rigorous decoupling using minimal interfaces we investigated six picked open source projects. We created a balanced test suite using popular Java projects[3] which span a number of domains.

JChessBoard [2] is a chess game capable using a regular TCP/IP connection to play against human opponents. Furthermore it is capable of editing and viewing the PGN[4] format.

JUnit [3] is a popular framework for unit testing in the Java programming language.

Mars [4] is a simple, extensible, services-oriented network status monitor written in Java.

GoGrinder [5] is a Java program for practicing Go problems using the SGF[5] format to load these problems.

DrawSWF [6] is a simple drawing application written in Java. The drawings created can be exported as an animated SWF (Macromedia Flash) file.

JHotDraw [7] is a well-known framework for developing two-dimensional structured drawing editors.

These projects have been completely refactored using InferType. Table 1 presents metrics regarding the size of the projects and decoupling before the refactoring occurs. We will discuss these results in detail in the next subsections.

3.2 Before the Refactoring

3.2.1 General Observations

In every project we found that there exist more classes than interfaces. Values range from about two classes per interface to about five classes per interface. We expect that after the refactoring the numbers are in favor of the interfaces, i.e. there are more interfaces than classes in each project.

Table 1 also reveals that there are developers, or project teams, which use interfaces for typing declaration elements, and those who don't. In particular in *JUnit* and *JHotDraw* more declaration elements are typed with interfaces than with classes. Contrary to these two projects a much smaller number of declaration elements are typed with interfaces in *Mars*, *GoGrinder* and *DrawSWF*. Even worse in *JChessBoard* there is not a single declaration element typed with an interface.

[2] Available at http://www.fernuni-hagen.de/ps/docs/InferType/.

[3] We used the popularity rating provided at http://www.freshmeat.net.

[4] PGN stands for "Portable Game Notation", a standard designed for the representation of chess game data using ASCII text files.

[5] SGF is the abbreviation of 'Smart Game Format'. The file format is designed to store game records of board games for two players.

As could be expected given the large number of available interfaces ACD values for both *JUnit* and *JHotDraw* are low. For example in *JHotDraw* a declaration element on average does not use 10% of the available methods, whereas a declaration element in *GoGrinder* on average does not use 57% of the available methods.

However, BCD values indicate that decoupling in all projects could be improved using only existing types. Nevertheless these improvements are small and therefore we conclude that developers already make good use of existing types for typing declaration elements.

3.2.2 The Projects in Detail

JChessBoard was the smallest project in our test suite. Even though half of the used types in these projects are interfaces, not a single declaration element is typed with an interface. This is due to the fact that JChessBoard extends classes from the JDK. These classes therefore contain methods from the JDK classes for which the formal parameters are typed with interfaces. Due to Java's typing rules the classes in JChessBoard have to implement these interfaces to make use of the inherited methods. Additionally 149 out of 190 declaration elements are typed with one out of five types from the total number of 49 available types. We expect that the benefit of refactoring in relation to the number of newly introduced types is biggest for this project.

JUnit is one of two projects in our test suite in which more declaration elements are typed with an interface than with a class. In *JUnit* the difference of the average ACD and the average BCD is significantly low, i.e. *JUnit*'s declaration elements are mostly typed with the best fitting type existing in the project. Furthermore the interface *junit.framework.Test* is used to type 68 out of 166 declaration elements. We expect that most of these declaration elements will be retyped with new interfaces, i.e. we expect that *junit.framework.Test* offers more methods than needed for most declaration elements.

Mars is the counterpart to *JUnit* regarding the usage of existing types. Redeclaration of every declaration element with existing types would already reduce the ACD value by 0,11. Notably, similar to *JChessBoard,* five out of 103 types account for 143 out of 278 declaration elements.

GoGrinder is similar to *JChessBoard* in terms of typing declaration elements with interfaces. Only 4% of all declaration elements in this project are typed with interfaces. Furthermore it has the highest average ACD value of all projects. We expect that some types, most likely the ones with a high ACD value, will trigger the creation of many new interfaces.

DrawSWF has the lowest class-to-interface ratio of all projects. There are approximately five times as many classes as interfaces used in this project. Furthermore half of the declaration elements were typed with 7% of the existing types.

JHotDraw is outstanding in two ways. First it is the project which makes most use of interfaces for typing declaration elements. Second both the average ACD value and the average BCD value are the lowest in our test suite, i.e. there is little coupling existing in this project and most times the best fitting and existing type is used to type a declaration element.

3.3 After the Refactoring

3.3.1 General Observations

Table 2 shows the same metrics as Table 1, but this time after using InferType on the projects. Note that we omitted all ACD and BCD values as the very purpose of the refactoring is making these values zero, in which it succeeded.

However, it is surprising that less new interfaces were introduced to the projects than one might fear. The worst case, i.e. one new interface for every existing declaration element, never occurred. Actually all projects were not even close to the worst case as the last row in Table 2 shows. This is an indication that at least some declaration elements are using the same access set and could therefore be declared with the same type. Nevertheless there are many newly introduced interfaces which are unpopular, i.e. there are only few declaration elements typed with these interfaces. Figures 1 to 8 in the appendix show the popularity, in terms of declaration elements typed with a particular interface, of each interface for every project.

Yet, not every declaration element is declared with an interface as its type. Except the two extremes *JChessBoard* and *JHotDraw* around two or three times as many declaration elements are typed with interfaces as with classes.

A comparison of the situation before and after the refactoring is show in Table 3. In the next section we will present more detailed information about the changes which occurred during the refactoring.

3.3.2 The Projects in Detail

JChessBoard profited the most from the refactoring which is not astonishing, because it was using no interface at all for typing declaration elements. About half of the declaration elements are retyped with interfaces during the refactoring. From 149 declaration elements declared with one out of five types only 73 declaration elements where still typed with these types after the refactoring. In particular all declaration elements formally typed with the inner class *STR* from *jchessboard.PGN* are now typed with an interface.

JUnit offered a little surprise as after the refactoring less declaration elements were typed with interfaces than before the refactoring. This is due to the fact that declaration elements formerly typed with an interface are now typed with *Object*, as the access set of these declaration elements was empty. To be precise, 42 out of 68 declaration elements of the interface *junit.framework.Test* are now typed with *Object.*

Mars had a similar starting position as *JChessBoard*. In both projects a small number of types have been used to type an overwhelming part of the existing declaration elements. Hence both projects behaved similar during the refactoring. Like in *JChessBoard*, from 143 declaration elements typed with one out of five types from all available types only 46 of them were still typed with these types after the refactoring. In particular from the 34 declaration elements typed with the clas *org.altara.mars.Status* only one was still typed with this class after the refactoring.

GoGrinder was the second worst project -*JChessBoard* being the worst- in terms of using interfaces for typing declaration elements. Furthermore it had the highest average ACD value of all projects, i.e. a declaration element in *GoGrinder* on average did not use 57% of the available methods, and we expected that some types will trigger the creation of many new interfaces. For example *GoGrinder.ProbCollection*, the most popular type before the refactoring, triggered the creation of 19 new interfaces for redeclaring the declaration elements formally typed with *GoGrinder.ProbCollection*.

DrawSWF had the lowest class-to-interface ratio before the refactoring and after the refactoring not much changed. It is interesting to note that one of the newly introduced interfaces is more popular than any interface or class before the refactoring[6]. This leads to the conclusion that this new interface is used to redeclare declaration elements from various types, a strong indication that an unwanted structural match occurred. This leads to circumstances in which two declaration elements are considered in terms of types, and therefore the methods which can be accessed, even though one of the objects is not. Thus even the program is type correct; semantics of the program might have changed. Furthermore 158 of the 253 newly introduced interfaces were so specific that each of them was used to retype only one declaration.

JHotDraw was the project which made the heaviest use of interfaces for typing declaration elements. Furthermore the ACD values were small throughout, i.e. the amount of unused methods was relatively small. Thus newly introduced interfaces are very specific. As a matter of fact, 164 out of 324 newly introduced interfaces were used to redeclare just one declaration element.

3.4 Costs of Rigorous Decoupling

About twice as many types exist after the refactoring than before refactoring in every project. Even though the introduction of additional types is necessary for removing unnecessary decoupling every additional type makes the type hierarchy harder to understand and maintain. For example the class *GoGrinder.ProbCollection* in the project *GoGrinder* implements as many as seventeen interfaces after the refactoring. Therefore to evaluate the refactoring we use the number of newly introduced types as the cost for the refactoring.

Table 3 shows the average reduction of the ACD value in relation to the number of new types introduced, i.e. the higher the value the better. We will use this number as our cost/benefit ratio as the number of new types is our cost of the refactoring, and the average reduction of the ACD value is the benefit of the refactoring.

The low number for *JHotDraw* is eye-catching but not surprising, as this project already used more interfaces for typing declaration elements than classes before the refactoring. As mentioned in the last section about half of the new interfaces were so specific that they were used to type just one declaration element each.

The cost/benefit ratio for *DrawSWF* is similar to the one of *JHotDraw*. This is due to the fact that during the refactoring many very specific interfaces have been introduced to this.

In the remaining four projects the newly introduced interfaces were not as specific as in the above mentioned projects. This circumstance is reflected in the higher values of the cost/benefit ratio. Still a big part of the newly introduced interfaces was so specific that only few declaration elements could be redeclared with these interfaces. Figures 1 to 8 in the appendix show the popularity of the newly introduced types for a project. It is eye-catching that all projects have a few popular and many unpopular types.

[6] Unfortunately we have to omit the data which provided this insight due to its length.

3.5 Popular Types

Therefore the most interesting insight we gained after refactoring for each project is that *popular access sets*, which lead to *popular interfaces* during the refactoring, exist in every project. Figures 1 to 8 in the appendix show the popularity of each inferred interface. The popularity of an interface is defined as the number of declaration elements (the y-axis in the figures) declared with this interface.

All the diagrams in Figures 1 to 8 suggest a pareto distribution [13]. As a matter of fact the distribution of declaration elements among the types approximately follows the 80/20 rule, i.e. 80% of all declaration elements are typed with 20% of the available types, whereas the remaining 20% of all declaration elements are typed with 80% of the available types. Unfortunately most areas in which such a distribution occurs suffer from the so called *long tail*. In our case the long tail are all those types which are used only by a few declaration elements.

The results *JHotDraw* provides strong evidence that the cost/benefit ration also suffers from this distribution and that popular interfaces should be preferred. The average ACD value of this project was already low before the refactoring, i.e. the declared types provided a good decoupling. The refactoring introduced many unpopular types which were used to retype just one declaration element. This consequently led to the worst cost/benefit ratio. We therefore conclude that using only the most popular types, i.e. the 20% which are used by 80% of the retyped declaration elements, instead of using minimal interfaces everywhere results in a better cost/benefit ratio in terms of average ACD decrease per type.

In [11] we presented a metric, a tool and a guideline for finding popular access sets for a specific type. Using the refactoring however explicitly declares interfaces for all popular access sets in a project. These popular interfaces can be introduced to the original version of the project to reduce the ACD value, yet keeping the number of new types limited.

4. LESSONS LEARNED

In [12] the author noted that interfaces represent the roles a class plays and vice versa. However using an automatic refactoring to introduce minimal interfaces for every declaration element violates this principle. For example all declaration elements typed with *junit.framework.Test* in *JUnit* obviously play a specific role which is designated by the name of the interface. After redeclaring these declaration elements with *Object* no indication to a role is left. In section 3.5 we have shown that popular types exist in every project after the refactoring. [11] has shown that in many cases a role can be found for these popular types.

Rigorous decoupling comes with a high cost as shown in section 3.4. In section 3.5 we argued that introducing only popular interfaces might significantly reduce coupling, yet keeping the number of new types small.

Finally the results from section 3.2 indicate that finding the best fitting and existing type in a project for typing a declaration element is not a problem. The difference of ACD and BCD value was low amongst all projects. This might be due to the fact that refactorings in prominent IDEs like Eclipse and IntelliJ exist which help the developer to find the best fitting type among all existing.

5. FUTURE WORK AND CONCLUSION

We have used an existing refactoring to evaluate both cost and benefit of the most rigorous decoupling as made possible by introducing context-specific types. Our results provide evidence that -as would be expected- rigorous decoupling is not a good idea. Too many unpopular interfaces are introduced during the refactoring. The data we have shown indicate that the best trade-off between decoupling and number of types is to introduce only the most popular interfaces for classes. We will have to adjust our refactoring and present data which either confirms or disproves our assumption.

6. ACKNOWLEDGMENTS

The author thanks Andreas Meißner and Phillip Mayer for implementing most of the described refactoring.

7. REFERENCES

[1] Löwy, J. *Programming .NET Components*, O'Reilly Media, 2005.

[2] JChessBoard is available from http://jchessboard.sourceforge.net

[3] JUnit is available from http://www.junit.org.

[4] Mars is available from http://leapfrog-mars.sourceforge.net/.

[5] GoGrinder is available from http://gogrinder.sourceforge.net/.

[6] DrawSWF is available from http://drawswf.sourceforge.net/.

[7] JHotDraw is available from http://www.jhotdraw.org/.

[8] Gamma, E. et al., *Design Patterns*, Addison-Wesley Professional, 1997.

[9] Dean, J. , Grove, D. and Chambers, C., *Optimization of object-oriented programs using static class hierarchy analysis*, In: *Proc of ECOOOP*, 1995, 77-101.

[10] Steimann, F., Mayer, P. and Meißner, A., *Decoupling classes with inferred interfaces*, In: *Proceedings of the 2006 ACM Symposium on Applied Computing*, (SAC) (ACM 2006).

[11] Forster, F., *Mining Interfaces In Java Programs*, Technical Report, Fernuniversität Hagen, 2006.

[12] Steimann, F., *Role = Interface: a merger of concepts*, Journal of Object-Oriented Programming 14:4, 2001, 23–32.

[13] Pareto Distribution, *http://en.wikipedia.org/wiki/Pareto_distribution*, last visit 27.05.2006

[14] Basili, V.R., Caldiera, G. and Rombach, D., *The goal question metric approach*, In: *Encyclopedia of Software Engineering*, (John Wiley & Sons, 1994).

Appendix A

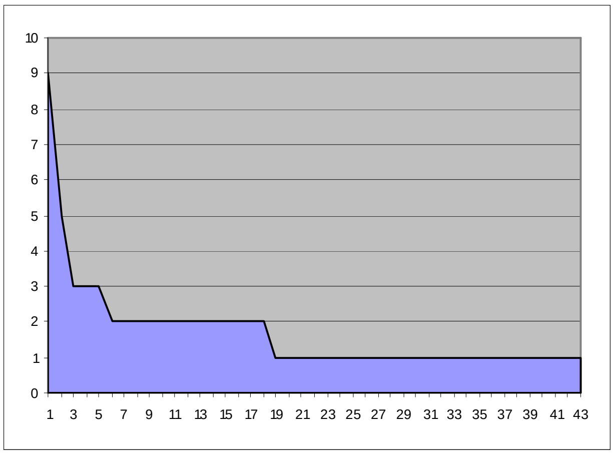


Figure 1: Popularity of new interfaces in JChessBoard

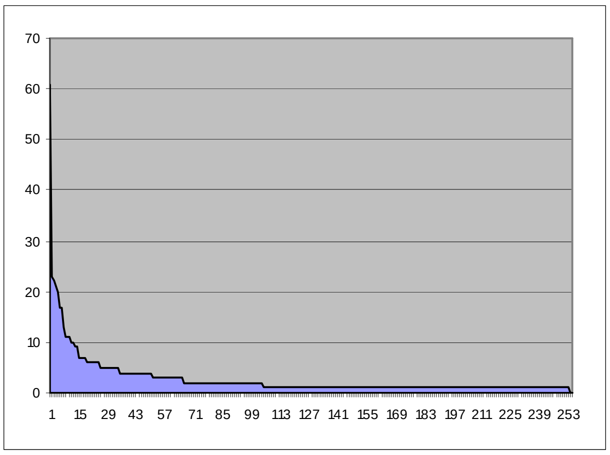


Figure 2: Popularity of new interfaces in DrawSWF

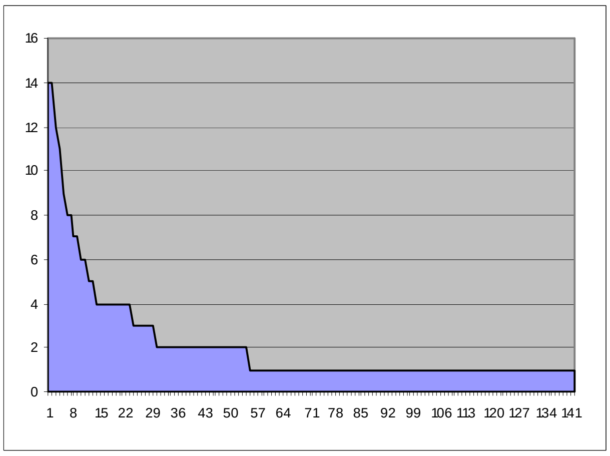


Figure 3: Popularity of new interfaces in GoGrinder

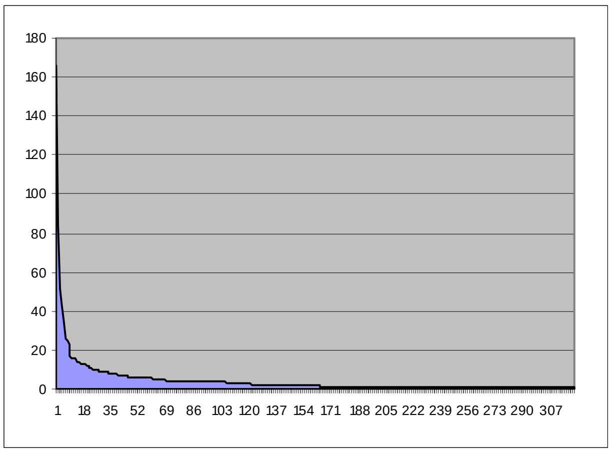


Figure 4: Interface Popularity of JHotDraw

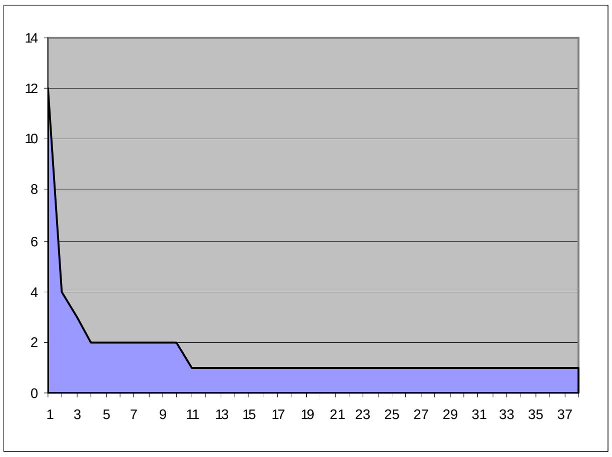


Figure 5: Interface Popularity of JUnit

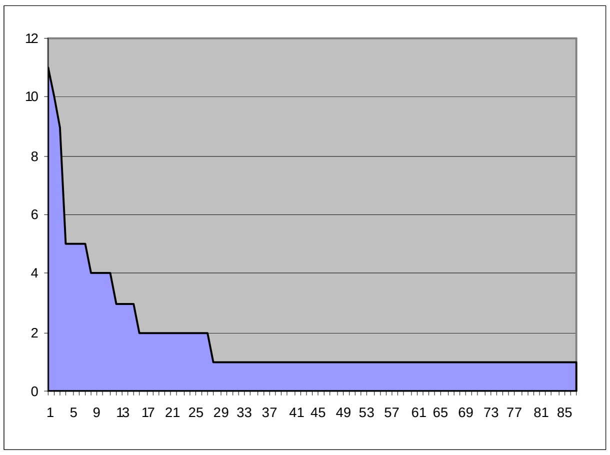


Figure 6: Interface Popularity of Mars

Dynamic Analysis of Program Concepts in Java

Jeremy Singer
School of Computer Science
University of Manchester, UK
jsinger@cs.man.ac.uk

Chris Kirkham
School of Computer Science
University of Manchester, UK
chris@cs.man.ac.uk

ABSTRACT

Concept assignment identifies units of source code that are functionally related, even if this is not apparent from a syntactic point of view. Until now, the results of concept assignment have only been used for static analysis, mostly of program source code. This paper investigates the possibility of using concept information as part of dynamic analysis of programs. There are two case studies involving (i) a small Java program used in a previous research study; (ii) a large Java virtual machine (the popular Jikes RVM system). These studies demonstrate the usefulness of concept information for dynamic approaches to profiling, debugging and comprehension. This paper also introduces the new idea of feedback-directed concept assignment.

1. INTRODUCTION

This paper fuses together ideas from program *comprehension* (concepts and visualization) with program *compilation* (dynamic analysis). The aim is to provide techniques to visualize Java program execution traces in a user-friendly manner, at a higher level of abstraction than current tools support. These techniques should enable more effective program comprehension, profiling and debugging.

1.1 Concepts

Program concepts are a means of high-level program comprehension. Biggerstaff et al [4] define a concept as 'an expression of computational intent in human-oriented terms, involving a rich context of knowledge about the world.' They argue that a programmer must have some knowledge of program concepts (some informal intuition about the program's operation) in order to manipulate that program in any meaningful fashion. Concepts attempt to encapsulate original design intention, which may be obscured by the syntax of the programming language in which the system is implemented. Concept *selection* identifies how many orthogonal intentions the programmer has expressed in the program. Concept *assignment* infers the programmer's intentions from the program source code. As a simple example, concept assignment would relate the human-oriented concept `buyATrainTicket` with the low-level implementation-oriented artefacts:

```
{   queue();
    requestTicket(destination);
    pay(fare);
    takeTicket();
    sayThankYou();
}
```

Often, human-oriented concepts are expressed using UML diagrams or other high-level specification schemes, which are far removed from the typical programming language sphere of discourse. In contrast, implementation-oriented artefacts are expressed directly in terms of source code features, such as variables and method calls.

Concept assignment is a form of reverse engineering. In effect, it attempts to work backward from source code to recover the 'concepts' that the original programmers were thinking about as they wrote each part of the program. This conceptual pattern matching assists maintainers to search existing source code for program fragments that implement a concept from the application. This is useful for program comprehension, refactoring, and post-deployment extension.

Generally, each individual source code entity implements a single concept. The granularity of concepts may be as small as per-token or per-line; or as large as per-block, per-method or per-class. Often, concepts are visualized by colouring each source code entity with the colour associated with that particular concept. Concept assignment can be expressed mathematically. Given a set U of source code units $u_0, u_1, \ldots, u_n$ and a set C of concepts $c_0, c_1, \ldots, c_m$ then concept assignment is the construction of a mapping from U to C. Often the mapping itself is known as the concept assignment.

Note that there is some overlap between concepts and aspects. Both attempt to represent high-level information coupled with low-level program descriptions. The principal difference is that concepts are universal. Every source code entity implements some concept. In contrast, only some of the source code implements aspects. *Aspects* encapsulate implementation-oriented cross-cutting concerns, whereas *concepts* encapsulate human-oriented concerns which may or may not be cross-cutting.

Throughout this paper, we make no assumptions about how concept selection or assignment takes place. In fact, all the concepts are selected and assigned manually in our two case studies. This paper concentrates on how the concept information is applied, which is entirely independent of how it is constructed. However we note that automatic concept selection and assignment is a non-trivial artificial intelligence problem.

1.2 Dynamic Analysis with Concepts

To date, concept information has only been used for static analysis of program source code or higher-level program descriptions [4, 10, 11]. This work focuses on *dynamic analysis* using concept information, for Java programs. Such dynamic analysis relies on embedded concept information within dynamic execution traces of programs.

1.3 Contributions

This paper makes three major contributions:

1. Section 2 discusses how to represent concepts practically in Java source code and JVM dynamic execution traces.

2. Sections 3.2 and 3.3 outline two different ways of visualizing dynamic concept information.

3. Sections 3 and 4 report on two case studies of systems investigated by dynamic analysis of concepts.

2. CONCEPTS IN JAVA

This section considers several possible approaches for embedding concept information into Java programs. The information needs to be apparent at the source code level (for static analysis of concepts) and also in the execution trace of the bytecode program (for dynamic analysis of concepts).

There are obvious advantages and disadvantages with each approach. The main concerns are:

- ease of marking up concepts, presumably in source code. We hope to be able to do this manually, at least for simple test cases. Nonetheless it has to be simple enough to automate properly.

- ease of gathering dynamic information about concept execution at or after runtime. We hope to be able to use simple dump files of traces of concepts. These should be easy to postprocess with perl scripts or similar.

- ease of analysis of information. We would like to use visual tools to aid comprehension. We hope to be able to interface to the popular Linux profiling tool Kcachegrind [1], part of the Valgrind toolset [16].

The rest of this section considers different possibilities for embedded concept information and discusses how each approach copes with the above concerns.

```
public @interface Concept1 { }
public @interface Concept2 { }
...
@Concept1 public class Test {
    @Concept2 public void f() { ... }
    ...
}
```

Figure 1: Annotation-based concepts in example Java source code

2.1 Annotations

Custom annotations have only been supported in Java since version 1.5. This restricts their applicability to the most recent JVMs, excluding many research tools such as Jikes RVM [2].

Annotations are declared as special `interface` types. They can appear in Java wherever a modifier can appear. Hence annotations can be associated with classes and fields within classes. They cannot be used for more fine-grained (statement-level) markup.

Figure 1 shows an example that uses annotations to support concepts. It would be straightforward to construct and mark up concepts using this mechanism, whether by hand or with an automated source code processing tool.

Many systems use annotations to pass information from the static compiler to the runtime system. An early example is the AJIT system from Azevedo et al [3]. Brown and Horspool present a more recent set of techniques [5].

One potential difficulty with an annotation-based concept system is that it would be necessary to modify the JVM, so that it would dump concept information out to a trace file whenever it encounters a concept annotation.

2.2 Syntax Abuse

Since the annotations are only markers, and do not convey any information other than the particular concept name (which may be embedded in the annotation name) then it is not actually necessary to use the full power of annotations. Instead, we can use *marker* interfaces and exceptions, which are supported by all versions of Java. The Jikes RVM system [2] employs this technique to convey information to the JIT compiler, such as inlining information and specific calling conventions.

This information can only be attached to classes (which reference marker interfaces in their `implements` clauses) and methods (which reference marker exceptions in their `throws` clauses). No finer level of granularity is possible in this model. Again, these syntactic annotations are easy to insert into source code. However a major disadvantage is the need to modify the JVM to dump concept information when it encounters a marker during program execution.

2.3 Custom Metadata

Concept information can be embedded directly into class and method names. Alternatively each class can have a special concept field, which would allow us to take advantage of the class inheritance mechanism. Each method can have a special concept parameter. However this system is thoroughly intrusive. Consider inserting concept information after the Java source code has been written. The concept information will cause wide-ranging changes to the source code, even affecting the actual API! This is an unacceptably invasive transformation. Now consider using such custom metadata at runtime. Again, the metadata will only be useful on a specially instrumented JVM that can dump appropriate concept information as it encounters the metadata.

2.4 Custom Comments

A key disadvantage of the above approaches is that concepts can only be embedded at certain points in the program, for specific granularities (classes and methods). In contrast, comments can occur at arbitrary program points. It would be possible to insert concept information in special comments, that could be recognised by some kind of preprocessor and transformed into something more useful. The Javadoc system supports custom tags in comments. This would allow us to insert concept information at arbitrary program points! Then we use a Javadoc style preprocessor (properly called a *doclet system* in Java) to perform source-to-source transformation.

We eventually adopted this method for supporting concepts in our Java source code, due to its simplicity of concept creation, markup and compilation.

The custom comments can be transformed to suitable statements that will be executed at runtime as the flow of execution crosses the marked concept boundaries. Such a statement would need to record the name of the concept, the boundary type (entry or exit) and some form of timestamp.

In our first system (see Section 3) the custom comments are replaced by simple `println` statements and timestamps are computed using the `System.nanoTime()` Java 1.5 API routine, thus there is no need for a specially instrumented JVM.

In our second system (see Section 4) the custom comments are replaced by Jikes RVM specific logging statements, which more efficient than `println` statements, but entirely non-portable. Timestamps are computed using the IA32 `TSC` register, via a new 'magic' method. Again this should be more efficient than using the `System.nanoTime()` routine.

In order to change the runtime behaviour at concept boundaries, all that is required is to change the few lines in the concept doclet that specify the code to be executed at the boundaries. One could imagine that more complicated code is possible, such as data transfer via a network socket in a distributed system. However note the following efficiency concern: One aim of this logging is that it should be unobtrusive. The execution overhead of concept logging should be no more than noise, otherwise any profiling will be inaccurate! In the studies described in this paper, the mean execution time overhead for running concept-annotated code is 35% for the small Java program (Section 3) but only 2% for the large Java program (Section 4).

```
// @concept_begin Concept1
public class Test {
    public void f() {
        ....

        while (...)
            // @concept_end Concept1
            // @concept_begin Concept2
    }
    ...
}
// @concept_end Concept2
```

Figure 2: Comments-based concepts in example Java source code

Figure 2 shows some example Java source code with concepts marked up as custom comments.

There are certainly other approaches for supporting concepts, but the four presented above seemed the most intuitive and the final one seemed the most effective.

3. DYNAMIC ANALYSIS FOR SMALL JAVA PROGRAM

The first case study involves a small Java program called `BasicPredictors` which is around 500 lines in total. This program analyses textual streams of values and computes how well these values could be predicted using standard hardware prediction mechanisms. It also computes information theoretic quantities such as the entropy of the stream. The program was used to generate the results for an earlier study on method return value predictability for Java programs [23].

3.1 Concept Assignment

The `BasicPredictors` code is an interesting subject for concept assignment since it calculates values for different purposes in the same control flow structures (for instance, it is possible to re-use information for prediction mechanisms to compute entropy).

We have identified four concepts in the source code, shown below.

system: the default concept. Prints output to stdout, reads in input file. Reads arguments. allocates memory.

predictor_compute: performs accuracy calculation for several *computational* value prediction mechanisms.

predictor_context: performs accuracy calculation for *context-based* value prediction mechanism (table lookup).

entropy: performs calculation to determine information theoretic entropy of entire stream of values.

The concepts are marked up manually using custom Javadoc tags, as described in Section 2.4. This code is transformed using the custom doclet, so the comments have been replaced by `println` statements that dump out concept information at execution time. After we have executed the instrumented program and obtained the dynamic execution trace which includes concept information, we are now in a position to perform some dynamic analysis.

3.2 Dynamic Analysis for Concept Proportions

The first analysis simply processes the dynamic concept trace and calculates the overall amount of time spent in each concept. (At this stage we do not permit nesting of concepts, so code can only belong to a single concept at any point in execution time.) This analysis is similar to standard function profiling, except that it is now based on specification-level features of programs, rather than low-level syntactic features such as function calls.

The tool outputs its data in a format suitable for use with the Kcachegrind profiling and visualization toolkit [1]. Figure 3 shows a screenshot of the Kcachegrind system, with data from the `BasicPredictors` program. It is clear to see that most of the time is spent in the `system` concept. It is also interesting to note that `predictor_context` is far more expensive than `predictor_compute`. This is a well-known fact in the value prediction literature [19].

3.3 Dynamic Analysis for Concept Phases

While the analysis above is useful for determining overall time spent in each concept, it gives no indication of the temporal relationship between concepts.

It is commonly acknowledged that programs go through different phases of execution which may be visible at the microarchitectural [7] and method [9, 15] levels of detail. It should be possible to visualize phases at the higher level of concepts also.

So the visualization in Figure 4 attempts to plot concepts against execution time. The different concepts are highlighted in different colours, with time running horizontally from left-to-right. Again, this information is extracted from the dynamic concept trace using a simple perl script, this time visualized as HTML within any standard web browser.

There are many algorithms to perform phase detection but even just by observation, it is possible to see three phases in this program. The startup phase has long periods of `system` (opening and reading files) and `predictor_context` (setting up initial table) concept execution. This is followed by a periodic phase of prediction concepts, alternately `predictor_context` and `predictor_compute`. Finally there is a result report and shutdown phase.

3.4 Applying this Information

How can these visualizations be used? They are ideal for visualization and program comprehension. They may also be useful tools for debugging (since concept anomalies often indicate bugs [22]) and profiling (since they show where most of the execution time is spent).

Extensions are possible. At the moment we only draw a single bar. It would be necessary to move to something resembling a Gantt chart if we allow nested concepts (so a source code entity can belong to more than one concept at once) or if we have multiple threads of execution (so more than one concept is being executed at once).

4. DYNAMIC ANALYSIS FOR LARGE JAVA PROGRAM

The second case study uses Jikes RVM [2] which is a reasonably large Java system. It is a production-quality adaptive JVM written in Java. It has become a significant vehicle for JVM research, particularly into adaptive compilation mechanisms and garbage collection. All the tests reported in this section use Jikes RVM version 2.4.4 on IA32 Linux.

A common complaint from new users of Jikes RVM is that it is hard to understand how the different adaptive runtime mechanisms operate and interact. So this case study selects some high-level concepts from the adaptive infrastructure, thus enabling visualization of runtime behaviour.

After some navigation of the Jikes RVM source code, we inserted concepts tags around a few key points that encapsulate adaptive mechanisms like garbage collection and method compilation. Note that all code not in such a concept (both Jikes RVM code and user application code) is in the default `system` concept.

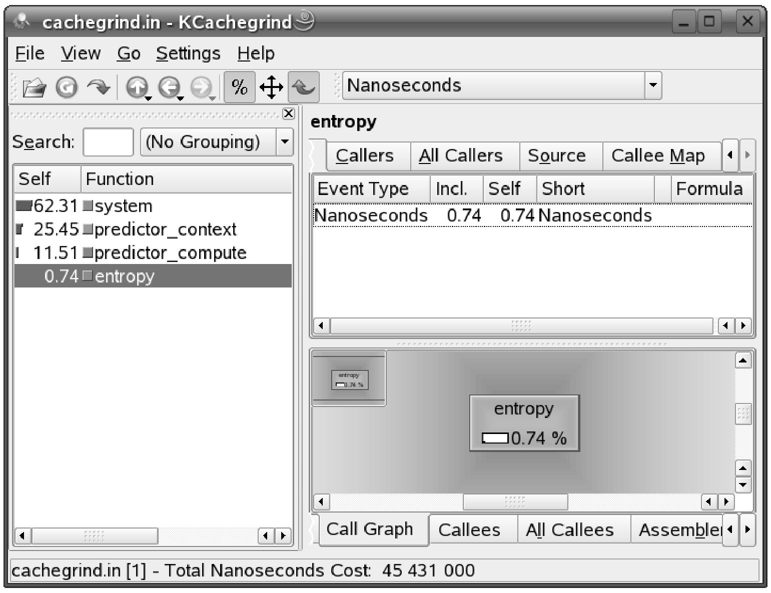
4.1 Garbage Collection

Figure 5 shows concept visualization of two runs of the `_201_compress` benchmark from SPEC JVM98. The top run has an initial and maximum heap size of 20MB (`-Xms20M -Xmx20M`) whereas the bottom run has an initial and maximum heap size of 200MB. It is clear to see that garbage collection occurs far more frequently in the smaller heap, as might be expected. In the top run, the garbage collector executes frequently and periodically, doing a non-trivial amount of work as it compacts the heap. In the bottom run, there is a single trivial call to `System.gc()` as part of the initial benchmark harness code. After this, garbage collection is never required so we assume that the heap size is larger than the memory footprint of the benchmark.

Many other garbage collection investigations are possible. So far we have only considered varying the heap configuration. It is also possible to change the garbage collection algorithms in Jikes RVM, and determine from concept visualizations what effect this has on runtime performance.

4.2 Runtime Compilation

In the investigation above, the `compilation` concept only captures optimizing compiler behaviour. However since Jikes RVM is an adaptive compilation system, it has several levels of compilation. The cheapest compiler to run (but one that generates least efficient code) is known as the *baseline* compiler. This is a simple macro-expansion routine from Java bytecode instructions to IA32 assembler. Higher levels of code efficiency (and corresponding compilation expense!) are provided by the sophisticated *optimizing* compiler, which can operate at different levels since it has many flags to enable or disable various optimization strategies. In general, Jikes RVM initially compiles application methods using the baseline compiler. The adaptive monitoring system identifies ‘hot’ methods that are frequently executed, and these are candidates for optimizing compilation. The hottest methods should be the most heavily optimized methods.

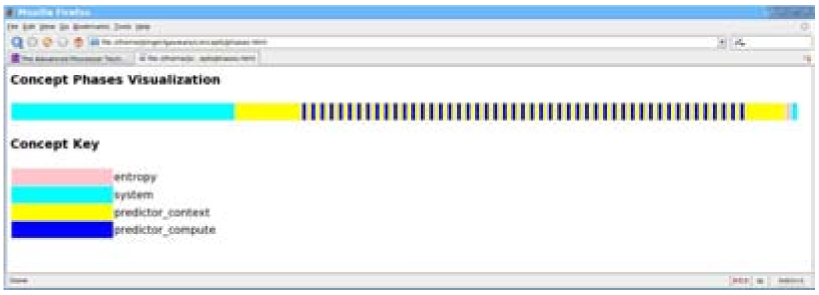


Self	Function
62.31	system
25.45	predictor_context
11.51	predictor_compute
0.74	entropy

Event Type	Incl.	Self	Short	Formula
Nanoseconds	0.74	0.74	Nanoseconds	

cachegrind.in [1] - Total Nanoseconds Cost: 45 431 000

Figure 3: Screenshot of Kcachegrind tool visualizing percentage of total program runtime spent in each concept



Concept Phases Visualization

Concept Key

🟪	entropy
🟦	system
🟨	predictor_context
🟦	predictor_compute

Figure 4: Simple webpage visualizing phased behaviour of concept execution trace

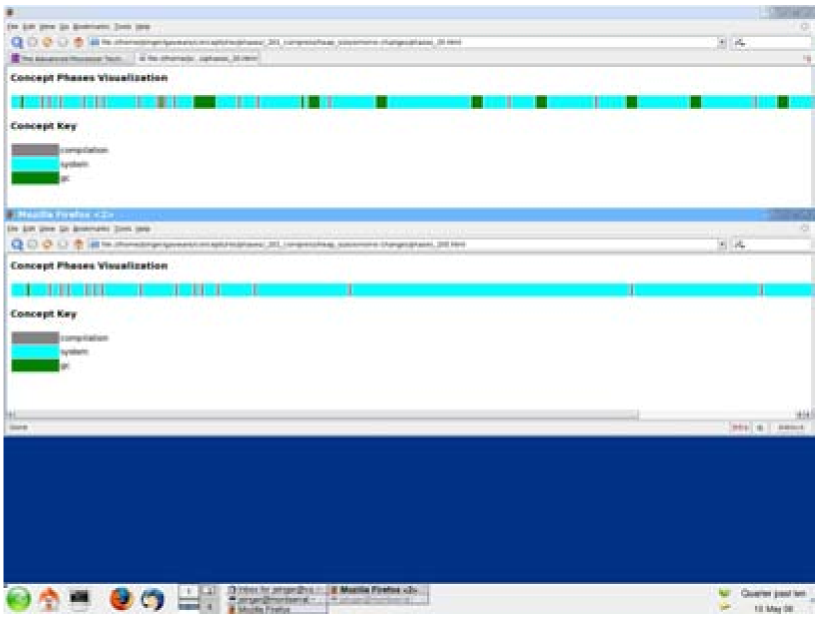


Figure 5: Investigation of garbage collection activity for different heap sizes

We use the `_201_compress` benchmark from SPEC JVM98 again. Figure 6 shows the start of benchmark execution using the default (adaptive) compilation settings (top) and specifying that the optimizing compiler should be used by default (bottom, `-X:aos:initial_compiler=opt`). In the top execution, the baseline compiler is invoked frequently for a small amount of time each invocation. On the other hand, in the bottom execution, the optimizing compiler is invoked more frequently. It takes a long time on some methods (since it employs expensive analysis techniques). However note that even with the optimizing compiler as default, it is still the case that there are some baseline compiled methods. This is not necessarily intuitive, but it is clear to see from the visualization!

Figure 7 shows the same execution profiles, only further on in execution time. The top visualization (default compilation settings) shows that many methods are now (re)compiled using the optimizing compiler. As methods get hot at different times, optimizing compiler execution is scattered across runtime. In the bottom visualization, once all the methods have been compiled with the optimizing compiler, there is generally no need for recompilation.

Note that both Figures 6 and 7 demonstrate that the optimizing compiler causes more garbage collection! The compilation system uses the same heap as user applications, and there is intensive memory usage for some optimizing compiler analyses.

There are plenty of other investigations to be performed with the Jikes RVM compilation system. In addition, we hope to identify other interesting concepts in Jikes RVM.

5. RELATED WORK

Hauswirth et al [13] introduce the discipline of *vertical profiling* which involves monitoring events at all levels of abstraction (from hardware counters through virtual machine state to user-defined application-specific debugging statistics). Their system is built around Jikes RVM. It is able to correlate events at different abstraction levels in dynamic execution traces. They present some interesting case studies to explain performance anomalies in standard benchmarks. Our work focuses on user-defined high-level concepts, and how source code and dynamic execution traces are partitioned by concepts. Their work relies more on event-based counters at all levels of abstraction in dynamic execution traces.

GCspy [18] is an elegant visualization tool also incorporated with Jikes RVM. It is an extremely flexible tool for visualizing heaps and garbage collection behaviour. Our work examines processor utilization by source code concepts, rather than heap utilization by source code mutators.

Sefika et al [20] introduce *architecture-oriented visualization.* They recognise that classes and methods are the base units of instrumentation and visualization, but they state that higher-level aggregates (which we term concepts!) are more likely to be useful. They instrument methods in the memory management system of an experimental operating system. The methods are grouped into architectural units (concepts) and instrumentation is enabled or disabled for each concept. This allows efficient partial instrumentation on a per-concept basis, with a corresponding reduction in the dynamic trace data size. Our instrumentation is better in that it can operate at a finer granularity than method-level. However our instrumentation cannot be selectively disabled, other than by re-assigning concepts to reduce the number of concept boundaries.

Sevitsky et al [21] describe a tool for analysing performance of Java programs using *execution slices.* An execution slice is a set of program elements that a user specifies to belong

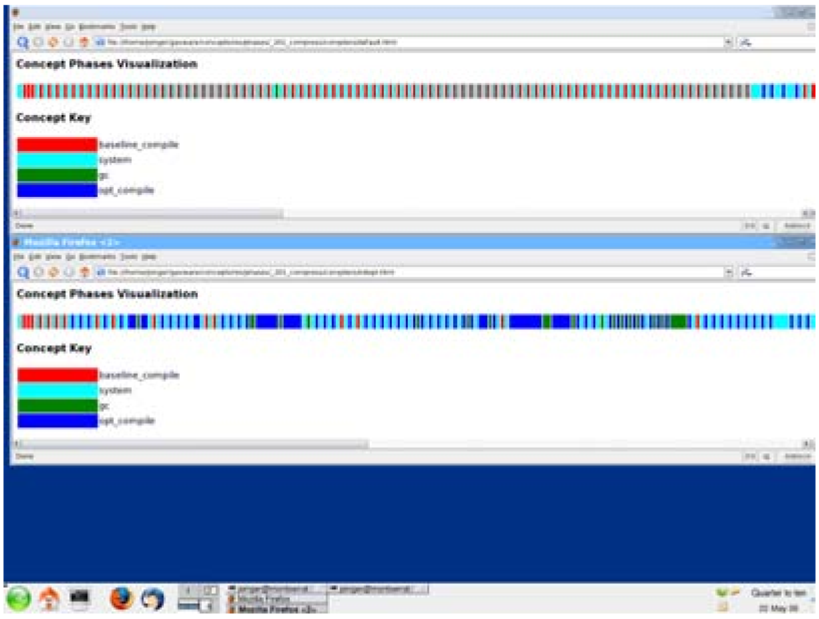


Figure 6: Investigation of initial runtime compilation activity for different adaptive configurations. Top graph is for default compilation strategy, i.e. use baseline before optimizing compiler. Bottom graph is for optimizing compilation strategy, i.e. use optimizing compiler whenever possible.

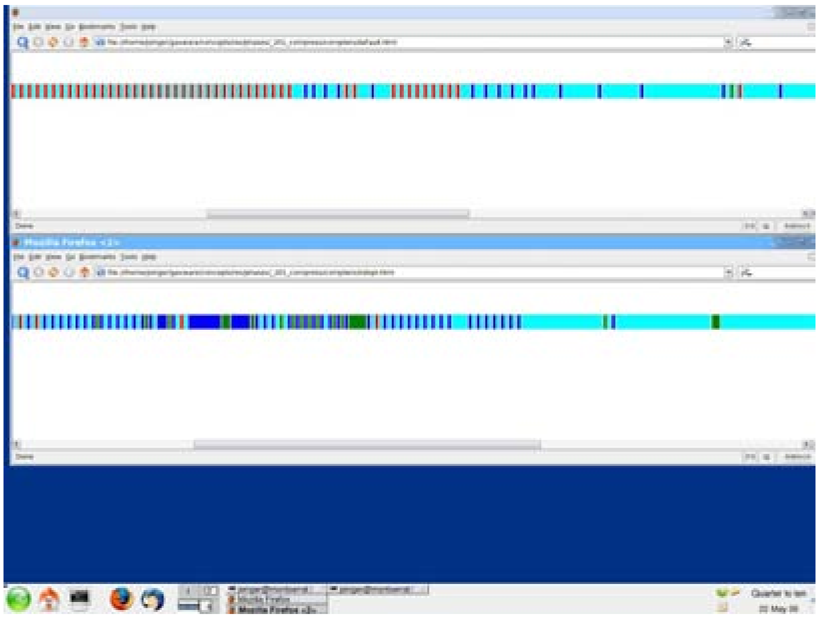


Figure 7: Investigation of later runtime compilation activity for different adaptive configurations. (Same configurations as in previous figure.)

to the same category—again, this is a disguised concept! Their tool builds on the work of Jinsight [17] which creates a database for a Java program execution trace. Whereas Jinsight only operates on typical object-oriented structures like classes and methods, the tool by Sevitsky et al handles compound execution slices. Again, our instrumentation is at a finer granularity. Our system also does not specify how concepts are assigned. They allow manual selection or automatic selection based on attribute values—for instance, method invocations may be characterized as slow, medium or fast based on their execution times.

Eng [8] presents a system for representing static and dynamic analysis information in an XML document framework. All Java source code entities are represented, and may be tagged with analysis results. This could be used for static representation of concept information, but it is not clear how the information could be extracted at runtime for the dynamic execution trace.

Other Java visualization research projects (for example, [6, 12]) instrument JVMs to dump out low-level dynamic execution information. However they have no facility for dealing with higher-level concept information. In principle it would be possible to reconstruct concept information from the lower-level traces in a postprocessing stage, but this would cause unnecessarily complication, inefficiency and potential inaccuracy.

6. CONCLUDING REMARKS

Until now, concepts have been a compile-time feature. They have been used for static analysis and program comprehension. This work has driven concept information through the compilation process from source code to dynamic execution trace, and made use of the concept information in dynamic analyses. This follows the recent trend of retaining compile-time information until execution time. Consider typed assembly language, for instance [14].

Feedback-directed concept assignment is the process of (1) selecting concepts, (2) assigning concepts to source code, (3) running the program, (4) checking results from dynamic analysis of concepts and (5) using this information to repeat step (1)! This is similar to feedback-directed (or profile-guided) compilation. In effect, this is how we made the decision to examine both baseline and optimizing compilers in Section 4.2 rather than just optimizing compiler as in Section 4.1. The process could be entirely automated, with sufficient tool support.

With regard to future work, we should incorporate these analyses and visualizations into an integrated development environment such as Eclipse. Further experience reports would be helpful, as we conduct more investigations with these tools. The addition of timestamps information to the phases visualization (Section 3.3) would make the comparison of different runs easier. We need to formulate other dynamic analyses in addition to concept proportions and phases. One possibility is *concept hotness*, which would record how the execution profile changes over time, with more or less time being spent executing different concepts. This kind of information is readily available for method-level analysis in Jikes RVM, but no-one has extended it to higher-level abstractions.

7. ACKNOWLEDGEMENTS

The first author is employed as a research associate on the Jamaica project, which is funded by the EPSRC Portfolio Award GR/S61270.

The small Java program described in Section 3 was written by Gavin Brown. We thank him for allowing us to analyse his program and report on the results.

Finally we thank the conference referees for their thoughtful and helpful feedback on this paper.

8. REFERENCES

[1] 2005. Kcachegrind profiling visualization, Josef Weidendorfer. see kcachegrind.sourceforge.net for details.

[2] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, 1999.

[4] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[5] R. H. F. Brown and R. N. Horspool. Object-specific redundancy elimination techniques. In *Proceedings of Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2006.

[6] P. Dourish and J. Byttner. A visual virtual machine for Java programs: exploration and early experiences. In *Proceedings of the ICDMS Workshop on Visual Computing*, 2002.

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, 2003.

[8] D. Eng. Combining static and dynamic data in code visualization. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–50, 2002.

[9] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 270–287, 2004.

[10] N. Gold. Hypothesis-based concept assignment to support software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 545–548, 2001.

[11] N. Gold, M. Harman, D. Binkley, and R. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software—Practice & Experience*, 35(10):977–1006, 2005.

[12] M. Golm, C. Wawersich, J. Baumann, M. Felser, and J. Kleinöder. Understanding the performance of the Java operating system JX using visualization techniques. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, page 230, 2002.

[13] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 251–269, 2004.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[15] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 27–33, 2004.

[16] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, 2003.

[17] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 151–162, 2002.

[18] T. Printezis and R. Jones. GCspy: an adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 343–358, 2002.

[19] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, 1997.

[20] M. Sefika, A. Sane, and R. H. Campbell. Architecture-oriented visualization. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 389–405, 1996.

[21] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *Proceedings of TOOLS Europe Conference*, 2001.

[22] J. Singer. Concept assignment as a debugging technique for code generators. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, 2005.

[23] J. Singer and G. Brown. Return value prediction meets information theory. In *Proceedings of the 4th Workshop on Quantitative Aspects of Programming Languages*, 2006. To appear in *Electronic Notes in Theoretical Computer Science.*

Investigating Throughput Degradation Behavior of Java Application Servers: A View from Inside a Virtual Machine

Feng Xian
Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
fxian@cse.unl.edu

Witawas Srisa-an
Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
witty@cse.unl.edu

Hong Jiang
Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
jiang@cse.unl.edu

ABSTRACT

Application servers are gaining popularity as a way for businesses to conduct day-to-day operations. Currently, the most adopted technologies for Application Servers are Java and .NET. While strong emphasis has been placed on the performance and throughput of these servers, only a few research efforts have focused on the degradation behaviors. Specifically, investigating how they perform under stress and factors that affect their throughput degradation behaviors. As a preliminary study, we conducted experiments to observe the throughput degradation behavior of Java application servers and found that the throughput degrades ungracefully. Thus, the goal of this work is three-fold: (i) identifying the primary factors that cause poor throughput degradation, (ii) investigating how these factors affect the throughput degradation, and (iii) observing how changes of algorithms and policies governing these factors affect the throughput degradation.

Categories and Subject Descriptors

D.3.4 [**Programming Language**]: Processors—*Memory management (garbage collection)*

General Terms

Experimentation, Languages, Performance

Keywords

Application servers, Throughput, Garbage collection

1. INTRODUCTION

Web applications have recently become a method of choice for businesses to provide services and gain visibility in the global market place. For example, eBay is enjoying over 180 million users worldwide. In addition, the advent of *Microsoft Office Live*[1], business productivity services, also propels the sophistication of Web services and applications to a new height. The enabling software that allows applications to be served through the Web is referred to as application servers. As of now, industrial observers expect applications servers to be a $27 billion business by the year 2010[2].

Two of the most adopted application server technologies are based on Java and .NET, which occupy about 70% of application servers market share (40% for Java and 30% for .NET) [16]. The major reason for such popularity is due to the rich set of libraries and features provided by these technologies that promote quick development and short time-to-market. On average, such technologies often reduce the code size and development cycle by 50% when compared to older technologies such as CGI written in C/C++[3].

Application servers often face significant variation in service demands — the higher demands often coincide with "the times when the service has the most value" [23]. Thus, these servers are expected to maintain responsiveness, robustness, and availability regardless of the changing demands. However, the current generation of application servers are not well equipped to meet such expectation as they often fail under heavy workload. For example, on the day that Apple announced the release its Video IPOD, the Apple Store site was down for over one hour due to heavy traffic[4]. In addition, these servers are susceptible to Distributed Denial of Service (DDoS) attacks. One notable example is when a group of Korean high school students launched a DDoS attack on a university Web site to prevent other students from applying[5]. While application server technologies continue to be widely adopted, the knowledge of why these servers fail and how to prevent them from failing is still elusive.

To date, very little research has been conducted on the throughput degradation behavior of Java application servers [9, 12]. Specifically, very little information is known about the systems behavior under stress. We have conducted experiments and found that the throughput of Java application servers does not degrade gracefully. A relatively small change in client's requests (22% increase) can cause the throughput to drop by as much as 75%. This is in contrast to the Apache Web server in which the throughput degrades much more gracefully. This paper reports the results of our extensive study to investigate the reasons behind poor throughput degradation in Java application servers. There are three major contributions resulting from our work.

[1]Microsoft Office Live, www.officelive.com

[2]"Web Services to Reach $21 Billion by 2007", Web Hosting News, http://thewhir.com/marketwatch/idc020503.cfm.

[3]Five Reasons to Move to the J2SE 5 Platform, January 2005, http://java.sun.com/developer/technicalArticles/J2SE/5reasons.html

[4]from netcraft.com, http://www.netcraft.com

[5]"Cyber Crime Behind College Application Server Crash" from http://english.chosun.com/w21data/html/news/200602/200602100025.html

PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany

1. Identifying the primary factors that cause the poor throughput degradation.
2. Investigating the effects of these factors on throughput.
3. Observing how changes of algorithms and policies in these factors affect the throughput degradation.

The remainder of this paper is organized as follows. Section 2 details our experiments to identify opportunities for improvement. Section 3 details our experimentation plan. Sections 4 and 5 report the results and discuss possible improvements. Section 6 discusses background information and highlights related work. Section 7 concludes the paper.

2. MOTIVATION

In 2001, Welsh *et al.* [23] reported three important trends that magnify the challenges facing Web-based applications. First, services are becoming more complex with widespread adoption of dynamic contents in place of static contents. Second, the service logics "tend to change rapidly". Thus, the complexity of development and deployment increases. Third, these services are deployed on general-purpose systems and not "carefully engineered systems for a particular service." Such trends are now a common practice. Complex services including entire suites of business applications are now deployed using Web application servers running commodity processors and open-source software. With this in mind, we conduct an experiment to observe the degradation behavior of Java application servers on an experimental platform similar to the current common practice (i.e. using Linux on X86 system with MySQL database and JBoss application server). For detailed information about the experimental setup, refer to section 3.2.

Initially, our experiments were conducted using the smallest amount of workload allowed by SPECjAppServer2004, a standardized benchmark to measure the performance of Java application servers. We set the maximum heap size to be twice as large as the physical memory—4 GB heap with 2 GB of physical memory in this case. We monitored the throughput delivered by the system. We then gradually increased the workload until the system refuses to service any requests.

For comparison, we also conducted another experiment to observe the degradation behavior of the Apache Web server (we used the same computer system and SPECweb2005 to create requested traffic). Since the two benchmarks report different throughput metrics — jobs per second for jAppServer2004 vs connections per second for Web2005 — we normalized the throughput and the workload to percentage. That is we considered the maximum throughput delivered by a system during an execution as 100% (referred to as *t*) and the maximum workload i.e. the workload that the systems completely refuse connection as 100% (referred to as *w*). The degradation rate (referred to as d) is $d = \frac{\Delta t}{\Delta w}$. The result of our comparison is shown in Figure 1.

The result shows that JBoss is able to deliver high throughput for about 60% of the given workload. However, when the workload surpasses 60%, the throughput reduces drastically. This system begins to refuse connection at 80% of the maximum workload. A drastic degradation in throughput (nearly 75%) occurs when the workload increases by only 20%. Thus, the degradation rate, *d*, is $\frac{0.75}{0.20} = 3.40$. Also notice that the value of *d* for the Apache is 1.69 (see Figure 1). A smaller value of *d* means that the application is more failure-resistant to increasing workload. We also investigated the effect of larger memory on the throughput. Again, larger memory improves the maximum throughput (see Figure 2 but has very little effect on the degradation behavior.

According to [12], the degradation behavior experienced in our experiments is considered ungraceful because such behavior can lead to non-robust systems. Moreover, it gives very little time to administer recovery procedures. The authors investigated the factors that affect throughput degradation behavior of Java Servlets by examining the operating system behaviors. They found that thread synchronization is the most prominent factor at the OS level. We would like to point out that their work did not study the factors within the Java virtual machine. On the contrary, our investigation concentrated specifically at the Java Virtual Machine level. Since Java Virtual Machines (*JVMs*) provide the execution environment for these application servers, we conjectured that the major factors that cause the throughput to degrade ungracefully reside in the Virtual Machines.

3. EXPERIMENTS

In this study, our main objectives are as follows:

Research Objective 1 (RO1): Identifying the major factors responsible for the rapidly declining throughput of Java application servers triggered by small workload increase.

Research Objective 2 (RO2): Investigating how these factors affect the throughput of Java server applications.

Research Objective 3 (RO3): Observing how the changes in algorithms and policies controlling these factors affect the throughput of Java application servers. To achieve this objective, we manipulate the algorithms and policies governing the behaviors of these factors.

3.1 Benchmarks

There are two major components in our experimental objects, the application servers and the workload drivers. The selected application servers must meet the following criteria. First, they must be representative of real-world/widely used application servers. Second, we must have accessibility to the source code to control and manipulate their execution context. Our effort began with the identification of server applications that fit the two criteria. We have investigated several possibilities and selected two open-source applications described below.

JBoss [13] is by far the most popular open-source Java application server (34% of market share and over five million downloads to date)[6]. It fully supports J2EE 1.4 with advanced optimization including object cache to reduce the overhead of object creation.

Java Open Application Server (JOnAS)[7] is another open-source application server. It is built as part of the ObjectWeb initiative. Its collaborators include the France Telecom, INRIA, and Bull (a software development company).

In addition to identifying the applications, we also need to identify workload drivers that create a realistic client/server environment. We choose an application server benchmark from SPEC, jAppServer2004 [19], which is the standard benchmark for testing the performance of Java application servers. It emulates an automobile manufacturing company and its associated dealerships. Dealers interact with the system using web browsers (simulated by a driver program) while the actual manufacturing process is accomplished via RMI (also driven by the driver). This workload stresses the ability of Web and EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc.

[6]from http://www.gridtoday.com/04/0927/103890.html

[7]JOnAS: Java Open Application Server available from http://jonas.objectweb.org

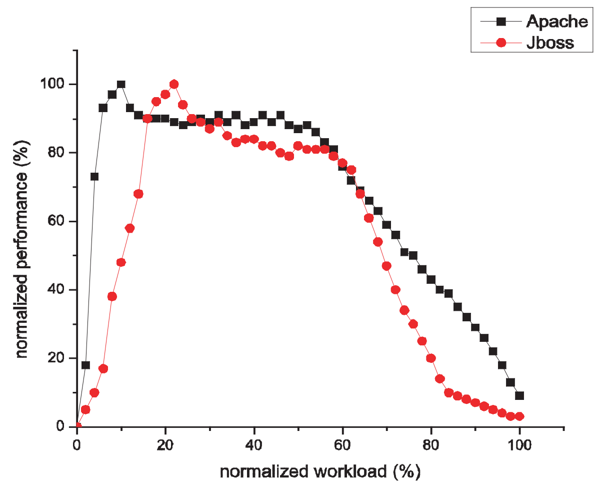


Figure 1: Throughput degradation behaviors of JBoss and Apache.

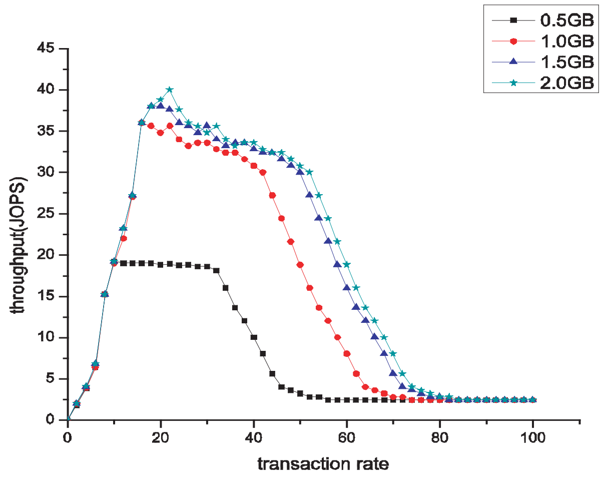


Figure 2: Throughput comparison with respect to heap sizes.

Workload of the benchmark is measured by transaction rate, which specifies the number of Dealer and Manufacturing threads. Throughput of the benchmark is measured by JOPS (job operations per second). The SPECjAppServer2004 Design Document [19] includes a complete description of the workload and the application environment in which it is executed.

3.2 Experimental Platforms

To deploy SPECjAppServer2004, we used four machines to construct the three-tier architecture. Since our experiments utilized both the Uniprocessor system and the Multiprocessor system, our configuration can be described as follows.

Uniprocessor application server (System A): The client machine is a dual-processor Apple PowerMac with 2x2GHz PowerPC G5 processors and 2 GB of memory. The server is a single-processor 1.6 GHz Athlon with 1GB of memory. The MySQL[8] database server is a Sun Blade with dual 2GHz AMD Opteron processors as the client machine running Fedora Core 2 and 2 GB of memory.

Multiprocessor application server (System B): The client machine is the same as the system above. However, we swapped the application server machine and the database server machine. Thus, the dual-processor Sun Blade is used as the application server, and the single-processor Athlon is used as the database server.

In all experiments, we used Suns J2SE 1.5.0 on the server side, and the young generation area is set to the default value, which is 1/9 of the entire heap and has shown to minimize the number of the expensive mature collections. We ran all experiments in standalone mode with all non-essential daemons and services shut down.

The virtual machine is instrumented to generate trace information pertaining to the runtime behavior such as object allocation information, reference assignment, execution thread information, and garbage collection (GC) information. It is not uncommon that such trace files be as large as several gigabytes. These trace files are then used as inputs to our analysis tool that performs lifetime analysis similar to the Merlin algorithm proposed by Hertz et al. [10]. The major difference between our approach and theirs is that ours uses off-line analysis and theirs uses on-line analysis. To obtain micro-architecture information, we utilize model specific performance monitoring registers.

[8]MySQL available from http://www.mysql.com

3.3 Variables and Measures

We utilized several workload configurations to vary the level of stress on the selected applications. In all experiments, we increased the workload from the minimum value available to the maximum value that still allow the application to operate. For example, we began our experiment by setting the workload value of SPECjAppServer2004 to 1. In each subsequent experiment, we increased the workload value until JBoss encounters failure. The failure point is considered to be the maximum workload that the system (combination of application server, JVM, OS, etc.) can handle. As shown in section 2, the throughput dramatically degrades as the workload increases. This degradation is likely caused by the runtime overhead. To address our RO1, we monitor the overall execution time (T), which is defined as:

$$T = T_{app} + T_{gc} + T_{jit} + T_{sync}$$

It is worth noticing that T_{app} is the time spent executing the application itself. T_{gc} is the time spent on garbage collection. T_{jit} is the time spent on runtime compilation. Many modern virtual machines use Just-In-Time (JIT) compilers to translate byte-code into native instructions when a method is first executed. This time does not include the execution of compiled methods; instead, it is the time spent on the actual methods compilation and code cache management. Finally, T_{sync} is the time spent on synchronization. We monitored synchronization operations such as lock/unlock, notify/wait, the number of threads yield due to lock contentions. We chose these time components because they have historically been used to measure the performance of Java Virtual Machines [2].

By measuring the execution of each run-time function, we can identify the function that is most sensitive to the increasing workload. The result of this research objective is used as the focal point in RO2. To address RO2, we further investigated the runtime behaviors of these factors. Once again, we varied the workload but this time, we also measured other performance parameters such as the number of page faults in addition to the throughput. These parameters give us more insight into the effect of these factors on the throughput. Specifically, we closely examined the governing policies of these runtime factors (causes) to gain more understanding of the effects they have on the throughput. To address RO3, we conducted experiments that adjust both the fundamental algorithms and the policies used by the runtime factors and observed their effects on the throughput. By making these changes, we expected to identify alternative algorithms and policies more suitable for Java application servers.

3.4 Hypotheses

We conjectured that increasing workload can affect two major runtime components of a JVM, threading and garbage collection. Our conjecture is based on two observations. First, increasing workload results in more simultaneous clients. This can, in turn result in more synchronization overhead, which affects performance. Second, higher workload also results in more object creations. Therefore, the heap gets filled up quicker; thus, garbage collection is called more frequently. In addition, a study has shown that the lifespan of objects in server applications is longer than that of objects in desktop applications [18]. Longer living objects can degrade the garbage collection efficiency, prolong garbage collection pause times, and reduce the throughput. We will conduct experiments to investigate the validity of these conjectures empirically based on the following hypothesis.

H1: Thread synchronization and garbage collection are the two run-time functions most sensitive to workload. Our second research question attempts to identify the causes that affect the performance of the identified runtime functions, and in turn, affect the throughput of the applications. We conjectured that runtime algorithms (e.g. generational garbage collection) and policies (e.g. when to call garbage collection) can greatly affect the performance of runtime functions. Therefore, our experiments are designed to also validate the following hypothesis.

H2: Runtime algorithms and management policies can affect the performance of runtime functions and overall throughput. Therefore, changes in the algorithms and/or policies can affect the throughput degradation behavior. We will conduct experiments to validate this hypothesis and report the preliminary results.

4. RESULTS

4.1 RO1: Factors that affect throughput.

We conducted experiments to identify factors that can affect the throughput of Java application servers. We measured the execution time of all major runtime functions in the virtual machine when the system is facing the lightest workload as well as the heaviest workload. Figure 3 reports the accumulated execution time (T). Notice that when the workload is light, only a small portion of time is spent in common VM functions. That is, T_{gc}, T_{sync}, and T_{jit} only account for 5%, 2% and 5% of the execution time, respectively. The remaining 88% is spent on application execution (T_{app}). Within this period, the maximum throughput is also achieved. Also notice that T_{jit} is very small and does not increase with workload. Because most commercial virtual machines do not discard compiled methods, they can be reused throughout the program execution [20, 27].

Synchronization as a factor. Work by [12] found that thread synchronization is the major factor that causes the throughput of the dynamic content generation tier to degrade differently among various operating systems; in most cases, the degradation was ungraceful. Because their observation was made mainly at the operating system level, it is not surprising for them to draw such a conclusion. Most runtime functions in JVMs may not utilize system calls. For example, memory allocators typically need system calls only when more space is needed in the heap. Thus, their methodology would regard runtime functions in the VM as application execution. We expect that more insight can be gained by observing the virtual

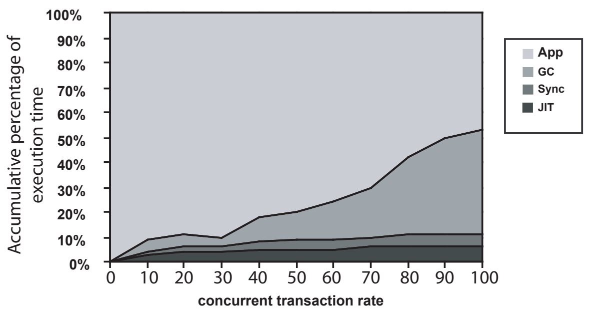


Figure 3: Accumulative time spent in major runtime functions.

machine performance.

As stated earlier, we monitored the accumulated execution time of all major runtime functions in the HotSpot VM. Our result is reported in Figure 3. Notice that the experimental result confirms our hypothesis that synchronization is workload sensitive as the time spent in synchronization becomes larger with increasing workload due to the increased resource contention. However, the increase is only a small percentage and should not affect the throughput degradation behavior.

Garbage Collection as a factor. On the other hand, we observed the amount of time spent in garbage collection as the workload increases. Figure 3 indicates that the time spent in garbage collection increases dramatically with heavier workload. Just prior to the complete failure of JBoss, the accumulative garbage collection time can take up over 50% of the overall execution time. We also found that garbage collection pauses can be as much as 300 seconds during the heaviest workload (see Table 1). As more time is spent on garbage collection, less time is spent on executing the application; thus, the throughput degrades dramatically. As a result, we conclude that garbage collection is a major factor that can affect the throughput and the degradation behavior of Java application server.

4.2 RO2: GC effects on throughput.

Currently, many commercial virtual machines including Sun J2SE 1.5 and Microsoft .NET CLR rely on generational garbage collection as the algorithm of choice for object management in server systems. Thus, our first focus will be on the effect of the generational algorithm on the application throughput. For more information on generational garbage collection, refer to Section 6.

Since the heap size can also affect the garbage collection performance (i.e. bigger heap translates to more time for objects to die), heap resizing policy can also play an important role. Thus, it is the second focus of our study.

4.2.1 *Effect of generational collector*

Generational collectors are designed to work well when the majority of objects die young. As reported earlier, the generational collector used in the JDK 1.5 performs extremely well when the workload is light. However, its throughput degrades significantly as the workload becomes much heavier. To understand the major causes of such a drastic degradation, we investigated the garbage collection frequency when the workload is heavy.

Notice that more time is spent on full collection as the workload is getting heavier (see Figure 4). At the heaviest workload, the system spent over 7000 seconds on full collection (about 36 times longer than that of minor collection and over 190 times longer than

Workload	Minor GC		Full GC	
	# of invocations	Avg. Pause (min:max) (seconds)	# of invocations	Avg. Pause (min:max) (seconds)
10	2037	0.021 (0.015:0.026)	48	0.78 (0.412:1.340)
20	2219	0.020 (0.011:0.033)	72	1.02 (0.232:2.021)
30	2901	0.022 (0.014:0.031)	115	1.13 (0.512:2.372)
40	3213	0.024 (0.011:0.039)	140	1.20 (0.412:3.721)
50	3907	0.021 (0.015:0.029)	192	1.45 (0.670:5.142)
60	4506	0.023 (0.012:0.026)	250	2.91 (1.010:7.020)
70	5102	0.027 (0.014:0.036)	370	3.31 (1.012:12.012)
80	5678	0.023 (0.015:0.037)	422	4.98 (2.102:34.014)
90	6150	0.025 (0.013:0.039)	512	6.12 (2.456:100.040)
100	7008	0.028 (0.015:0.039)	709	10.10 (3.124:300.024)

Table 1: Garbage collection activities and pause times.

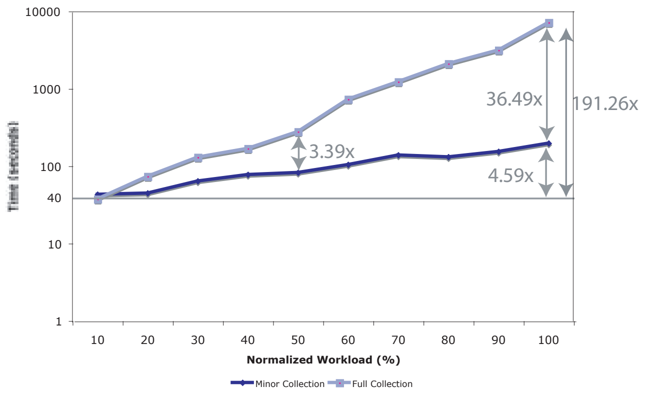


Figure 4: Time spent in minor and full collection.

the time spent on full GC at the lightest workload). We further investigated this behavior and found that frequent garbage collection prevents application from making any significant progress. In effect, the garbage collector simply thrashes. Thus, mark-sweep collector used for full collection would touch objects again and again. Therefore, the garbage collection processing cost becomes very high. As more time is spent on garbage collection, less time is spent executing the server applications.

Remote versus Local Objects. A study by [18] has shown that there are commonly two types of objects in .NET server appliations, *local* and *remote*. Remote objects are defined as objects created to serve remote requests [9], and local objects are created to serve local requests [10]. The study showed that objects of these two corresponding types have very distinct lifespan, remote objects tend to live much longer. To investigate whether such observation applies to our experiments, we observed the amount of remote objects and local objects over the allocation time during the heaviest workload and the their lifespans. Figure 5 report our findings.

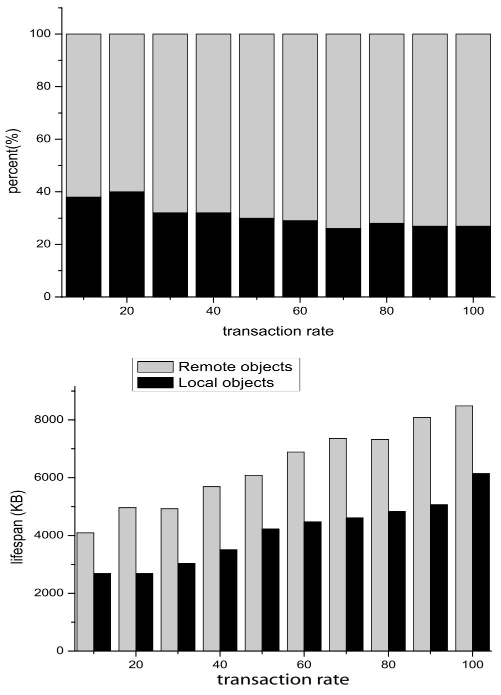


Figure 5: Comparison of quantity and lifespans of local and remote objects.

The figures indicate the ratio of these two types of objects in the heap and their average lifespan in one execution of jAppServer2004 running on JBoss. Figure 5 shows that remote objects dominate the heap in all the workload settings. Moreover, their average lifespan is longer than that of local objects, as indicated in Figure 5. Thus, the remote objects in JBoss are mostly long-lived and generational garbage collector will likely spend additional time and resources to promote these long-lived remote objects.

4.2.2 *Effect of heap enlargement policy*

We also investigated heap enlargement policy as a possible cause of poor garbage collection performance. Typically, there are two major considerations to perform heap resizing: when to resize and by how much.

In HotSpot VM, the pause time of each GC is evaluated by ap-

[9]We define remote object as a remotable object (a remotable object is the object which implements interface java.rmi.Remote) or an object rooted in a remotable object. In Java Application Servers (J2EE, JBoss, etc), all EJBs are remotable objects. They implement two interfaces: EJBHome and EJBObject, which extend the superinterface java.rmi.Remote.

[10]all objects other than remote objects are local objects

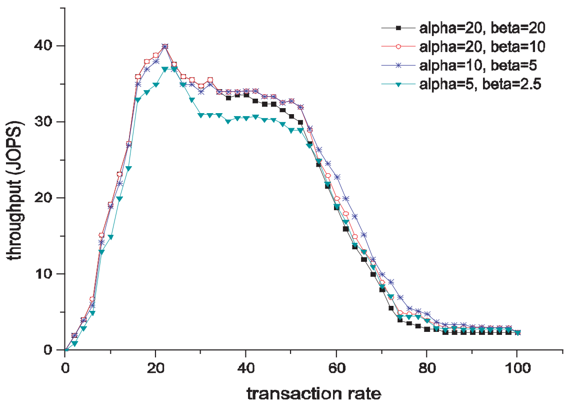


Figure 7: Throughput after applying the adaptive sizing mechanism.

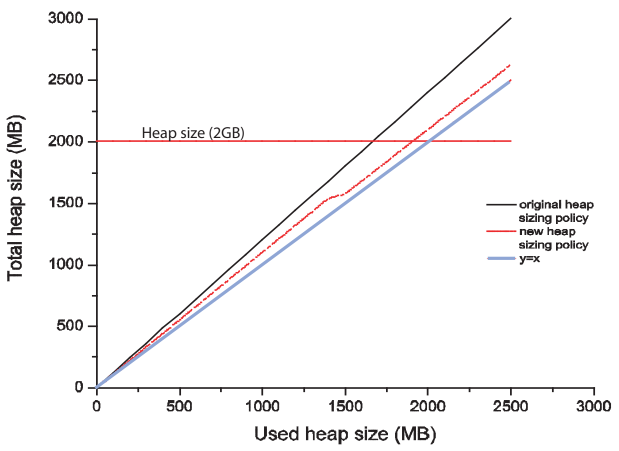


Figure 8: Heap usage with the adaptive sizing mechanism.

plying a linear curve fitting process on recent GC pause times. If the slope of the resultant linear function is positive, which means GC pause time is increasing, the VM will expand the heap by 20%. If the slope is negative, then the system will reduce the heap by 4%. This approach had two significant drawbacks based our experimental observation. First, the VM increases heap aggressively but reduces the heap too conservatively. When the footprint in the heap is smaller than the memory size but the total heap is larger than the memory size, it takes a long time to reduce the heap.

Second, the heap enlargement mechanism does not take into account the physical memory available and often indiscriminately grows larger than physical memory very quickly. For example, Figure 6 (right) shows the heap sizing activity at the heaviest workload (transaction 100). Note that the physical memory size is 2GB. The solid line is the new heap size after each sizing point. The dotted line is the actual heap usage (i.e., the amount of live objects) after a GC invocation. The star line is the number of page faults during the lifetime measured using the scale shown on the left y-axis. The figure shows that the heap is increased to be larger than the physical memory size at about 33% of the execution time. At this point, the actual heap usage is still smaller than the physical memory size. This enlargement induces a large amount of page faults for the remainder of the execution. As stated earlier (see Table 1), the pause time can be as long as 300 seconds as a significant amount of page faults occur during a full collection.

Remark. We found that generational garbage collection may not be suitable for application server under stress. This is due to a large number of objects in applications servers tend to be longer living; thus, less objects are collected in each minor collection and more full collection is invoked.

In addition, the current policy enlarges the heap very frequently to yield optimal garbage collection performance early on. In this strategy, the heap can become so large that the heap working set can no longer fit within the physical memory capacity. If this point is reached too soon, the system would spend a large amount of time servicing page faults. This is especially true during the mature collection, as mark-sweep has been known to yield poor paging locality [14, 11]. Two possible solutions to address this issue are (i) to use garbage collection techniques that are more suitable for long-lived objects and (ii) to adaptively resize the heap based on the amount of physical memory and transaction time requirement. We will preliminarily evaluate these two options in the next subsection.

4.3 RO3: Effects of changes in algorithms and policies on throughput.

4.3.1 Adaptive Heap Sizing Mechanism

From the previous section, we discovered that the adopted policy in the HotSpot VM increases the heap size so quickly that the heap exceeds the physical memory capacity very early in the execution. As a result, the system suffers a large number of page faults. We experimented with a new adaptive heap sizing policy that has been implemented into the HotSpot. Our new policy attempted to maintain the lowest possible heap size especially when the physical memory resource is scarce. As stated earlier, there are two considerations to perform heap resizing: when to expand or reduce the heap and by how much.

Our approach does not change the decision of when to resize the heap. However, we changed the adjustment quantity. Based on our study of the current heap sizing policy in the HotSpot, we noticed that page faults begin to occur when the heap is larger than 75% of physical memory (e.g. 1500 MB heap in a system with 2 GB physical memory). Thus, we used this insight to set a parameter to adjust our sizing policy. Basically, our threshold value is 75% of physical memory. When the current heap size is smaller than the threshold, the heap is increase by α percent during an expansion. Once the heap size exceeds the 75% threshold, we then reduced the percentage of enlargement to β percent ($\beta < \alpha$). We investigated the throughput and its degradation behavior under four different configurations of α and β: α=20/β=20, α=20/β=10, α=10/β=5, and α=5/β=2.5 Notice that α=20 and β=20 represents the original policy. In the α=20/β=20 approach, the heap is always increased by 20% of the current size, no matter if it exceeds the physical memory capacity or not. In our adaptive approach, the JVM adjusts the increasing percentage according to the available memory space. For example, in the α=10/β=5 approach, the heap is enlarged by 10% prior to 1500 MB heap size; afterward, the heap is increased by only 5%.

Figure 7 reports our finding. It is worth noticing that the changes in heap sizing policy have only minor effect on the throughput and degradation performance. However; conservative growth policy can significantly degrade the throughput as shown with α=5/β=2.5 configuration. Also notice that the current policy used by the HotSpot VM (α=20/β=20) does not yield the best throughput, instead, α=10/β=5

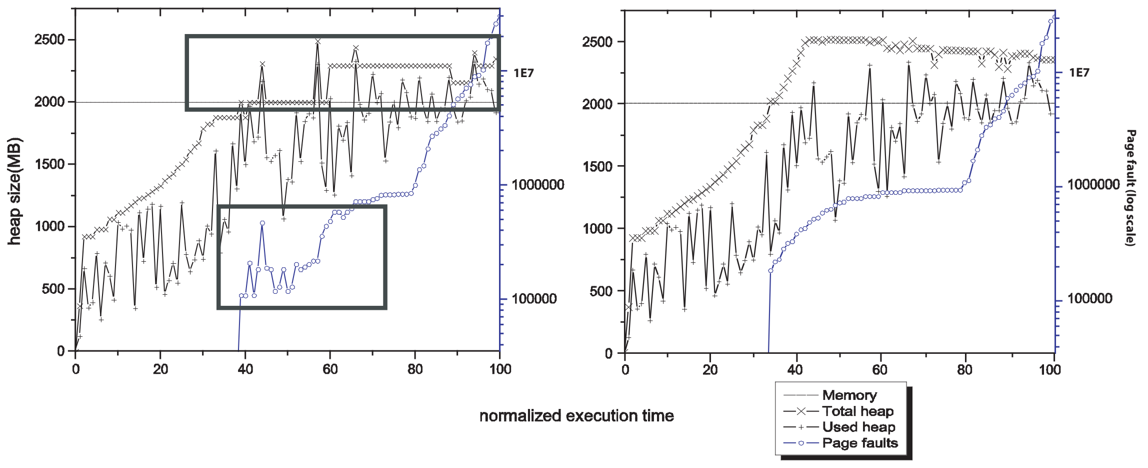


Figure 6: Comparing memory and paging activities of Jboss with (left) and without (right) the adaptive sizing mechanism (transaction rate=100).

yields the best throughput throughout the execution. Even though the proposed adaptive heap sizing policy has very little effect on the throughput degradation behavior, it can yield two additional benefits: lower heap usage and smaller number of page faults.

Reduction in heap usage. In Figure 8, we compared the amount of heap space needed by the application server with the actual heap size allocated by the JVM using two policies: α=20/β=20 and α=10/β=5. As a reminder, α=20/β=20 is the approach used in the current HotSpot VM and α=10/β=5 has shown to yield higher throughput. Notice that the proposed adaptive heap sizing policy utilizes the heap space more efficiently by committing the heap memory only slightly higher than the actual heap usage (125 MB). On the other hand, the approach currently used by the HotSpot committed a much larger amount of additional memory (about 500 MB) once the memory usage exceeds the physical memory.

Reduction in page faults. We compared the number of page faults between the two policies: α=20/β=20 and α=10/β=5. Figure 6 shows that our decision to slow the growth percentage at the beginning (α=10 instead of α=20) results in a reduction in the number of page faults early on (indicated by the rectangular box). The reduction is about 10%. However, the further reduction after the threshold is reached has very little effect on the number of page faults. Based on our results, we conclude that:

- Moderately conservative heap sizing policy only has slight effect on the maximum throughput. This is illustrated when we can achieve the best throughput with α=10/β=5 approach.
- Moderately conservative heap sizing policy can significantly reduce the number of page faults. However, the technique is more effective before the threshold is reached.
- Moderately conservative heap sizing policy can reduce the amount of memory usage and slightly improves the throughput throughout the execution. However, it has very little effect on the throughput degradation behavior.

4.3.2 *Improving Garbage Collection Paralellism*

Starting in J2SE 1.4.x, Sun also provides two additional GC techniques, parallel garbage collection (ParGC) and concurrent garbage collection in addition to the default generational mark-sweep [9].

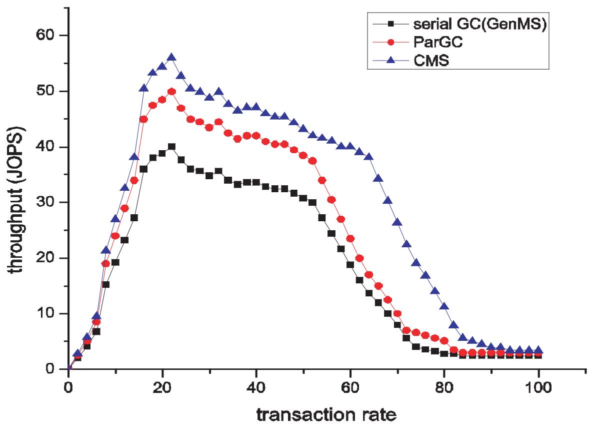


Figure 9: Effect of CMS on throughput.

The parallel collector is similar to the generational mark-sweep approach except the it utilizes parallel threads to perform minor and major collection. Thus, it is a stop-the-world approach designed to minimize pause time.

In Concurrent Mark-Sweep (CMS), a separate garbage collector thread performs parts of the major collection concurrently with the applications threads. For each major collection the concurrent collector will pause all the application threads for a brief period at the beginning of the collection and toward the middle of the collection. The remainder of the collection is done concurrently with the application. This collector performs parallel scavenging in the minor collections and concurrent mark-and-sweep in the major collections.

According to Sun, the Concurrent collector is ideal for server applications running on multi-processor systems (it cannot be used in single-processor systems). A study by Sun has shown that the concurrent collector can deliver higher throughput than the other approaches. However, the effects of the concurrent garbage collector on the throughput degradation behavior are not known. Therefore, the goal of this experiment is to investigate the effect of CMS on throughput degradation behavior. Note that we used *system B*, the

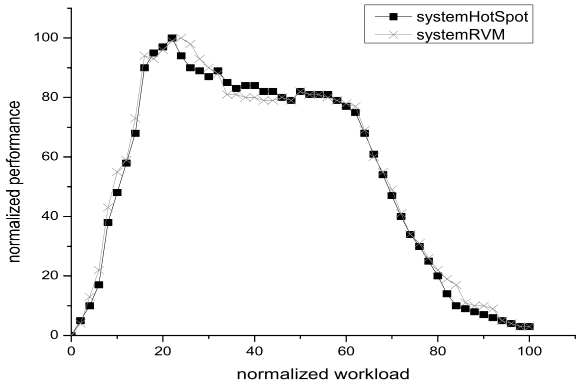


Figure 10: Comparing throughputs of $system_{HotSpot}$ and $system_{RVM}$.

multi-processor system for this experiment.

Figure 9 clearly indicates that CMS can greatly improve the maximum throughput of the system. The difference in throughputs between the concurrent collector and the single threaded generational mark-sweep (GenMS) can be as high as 40%. However, comparing the degradation rates of the three GC techniques, d_{CMS}, d_{ParGC}, and d_{genMS}, shows that both CMS and ParGC did't exhibit significant different on degradation rates. Based on this finding, we concluded that *the concurrent collector running on a more powerful computer system improves the maximum throughput due to better parallelism, but does not improve the throughput degradation behavior*.

4.3.3 Different Garbage Collection Techniques

We conducted our experiments on the Jikes RVM due to its flexibility in choosing multiple garbage collection algorithms. Since JBoss is not supported on the RVM, we also need to use a different application server. JOnAS is another open-source application server that is supported by the RVM. Once again, we use SPECjAppServer2004 as the workload driver.

To make certain that our substitution still provides a sound experimental platform, we conducted an experiment to compare the throughput degradation behaviors of the two systems, $system_{HotSpot}$ (SPECjAppServer running on JBoss and J2SE 1.5) and $system_{RVM}$ (SPECjAppServer running on JOnAS and RVM using generational collection (GenMS)). If the two systems show similar throughput pattern (based on normalized information), we assumed that any improvements resulting from modification of $system_{RVM}$ would also translate to similar improvements in $system_{HotSpot}$ if similar modifications were also applied. Figure 10 depicts the results of our comparison. Notice that the patterns are nearly identical.

Next, we conduct a set of experiments using different garbage collection techniques. The goal of these experiments is to compare the differences in the throughput behavior of each technique from the reference configuration ($system_{RVM}$). The description of each technique is given below.

GenMS: This hybrid generational collector uses a copying nursery and the MarkSweep policy for the mature generation. It is very similar to the generational mark-and-sweep collector in HotSpot. Thus, it is used as the reference configuration.

SemiSpace: The semi-space algorithm uses two equal sized copy spaces. It contiguously allocates into one, and reserves the other space for copying into since in the worst case all objects could survive. When full, it traces and copies live objects into the other space, and then swaps them.

GenCopy: The classic copying generational collector [1] allocates into a young (nursery) space. The write barrier records pointers from mature to nursery objects. It collects when the nursery is full, and promotes survivors into a mature semi-space. When the mature space is exhausted, it collects the entire heap.

MarkSweep: It is a tracing and non-generational collector. When the heap is full, it triggers a collection. The collection traces and marks the live objects using bit maps, and lazily finds free slots during allocation. Tracing is thus proportional to the number of live objects, and reclamation is incremental and proportional to allocation.

RefCount: The deferred reference-counting collector uses a freelist allocator. During mutation, the write barrier ignores stores to roots and logs mutated objects. It then periodically updates reference counts for root referents and generates reference count increments and decrements using the logged objects. It then deletes objects with a zero reference count and recursively applies decrements. It uses trial deletion to detect cycles [3, 4].

GenRC: This hybrid generational collector uses a copying nursery and RefCount for the mature generation [7]. It ignores mutations to nursery objects by marking them as logged, and logs the addresses of all mutated mature objects. When the nursery fills, it promotes nursery survivors into the reference counting space. As part of the promotion of nursery objects, it generates reference counts for them and their referents. At the end of the nursery collection, GenRC computes reference counts and deletes dead objects, as in RefCount.

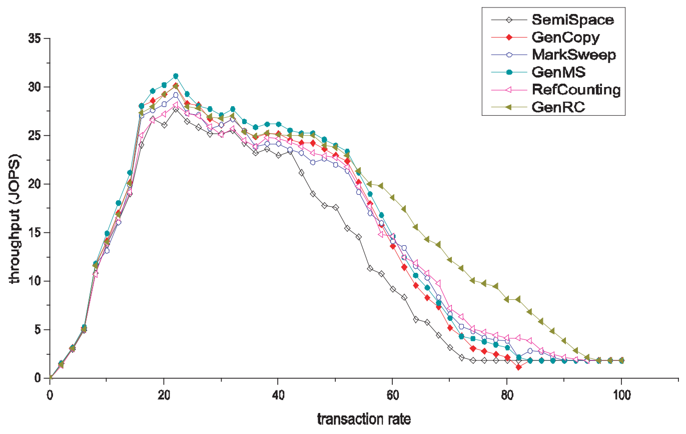


Figure 11: Comparing the throughputs of different GC techniques.

Figure 11 reports our finding. It is worth noticing that most techniques yield very similar throughput degradation behaviors. The two exceptions are SemiSpace and GenRC. For SemiSpace, the collection time is proportional to the number of live objects in the heap. Its throughput suffers because it reserves a half of heap for copying and repeatedly copies objects that survive for a long time. Additionally, its throughput performance and its responsiveness suffer because it collects the entire heap every time. Therefore, it has the lowest throughput at all workloads compared to the other 5 collectors.

GenRC on the other hand, allows the throughput of the application server to degrade much more gracefully. Unlike GenMS in which mature collection is frequently invoked during the heaviest workload, GenRC allows mature collection to be performed incrementally; thus, long pauses are eliminated and the memory space is recycled more efficiently. In addition, GenRC also ignores mutations of the young objects; thus, the overhead of reference manipulations is avoided.

5. DISCUSSION

In Java application servers, objects can be classified into local objects and remote objects, depending on the type of services they were created for. We have demonstrated that remote objects tend to be long-lived. We are currently working on a Service-Oriented garbage collection that segregates objects based on service types. The simulation results have shown that the scheme can significantly reduce the number of mature collection invocations.

In this paper, we investigated GC overhead by looking at several GC components like heap size, collector algorithm, triggering mechanism and sizing policy. We did not consider other non-GC components in the VM that may influence GC and server throughput. One possible component is thread scheduling. In the multi-threaded server environment, a large number of threads are created and operate simultaneously. We plan to experiment with thread scheduling algorithms that are GC cognizant. For example, threads that are expected to relinquish a large amount of memory may be given a higher priority so that memory is timely reclaimed. We expect that such algorithms can further improve throughput and affect the degradation behavior.

6. BACKGROUND AND RELATED WORK

6.1 Garbage Collection Overview

One of the most useful language features of modern object-oriented programming languages is garbage collection (GC). GC improves programming productivity by reducing errors resulting from explicit memory management. Moreover, GC underpins sound software engineering principles of abstraction and modularity. GC leads to cleaner code since memory management concerns are no longer cluttered with the programming logic [6, 14]. For our purpose, we summarize three garbage collection schemes that are related to this paper: mark-sweep, generational, and reference counting. Refer to [14, 24] for comprehensive summaries of the collection algorithms.

Mark and sweep collection [15] consists of two phases: *marking* and *sweeping*. In marking phase, the collector distinguishes live objects by tracing the heap and marking each of the live objects found. Tracing is basically traversing all live references to heap memory, to find all of the reachable objects. The traversal usually starts from a set of *roots*—such as program stacks, statically allocated memory, and registers—and results in a transitive closure over the set of live objects. In sweeping, the memory is exhaustively examined to find all the unmarked (garbage) objects and the collector "sweeps" their space by linking them into a free list. After sweeping, the heap can be compacted to reduce fragmentation.

Generational garbage collection [14, 21] segregates objects into "generations" using age as the criterion. The generational collection exploits the fact that objects have different lifetime characteristics; some objects have short lifespan while others live for a long time. Distribution wise, studies have shown that "most objects die young" (referred to as the weak generational hypothesis [21, 14]). Thus, the main motivation is to collect in the youngest generation, which is only a small portion of the heap, as frequently as possible. Collection in the young generation is referred to as *minor* and collection of the entire heap is referred to as *major* or *full*. Since most of the generational collectors are copying based (refer to [14] for more information), small numbers of surviving objects translate to short garbage collection pauses because there are less number of objects to traverse and copy. Short pauses often translate to higher *efficiency*—a ratio of the space collected over the space that has been used prior to a collection [14, 17].

Reference Counting (*RC*) [22] records the number of references to an object in its reference count field (often resides in the object's header [7, 14]). The counter is initialized to zero when the object is allocated. Each time a pointer to that object is copied, the reference count field is incremented, and each time a pointer to that object is removed, the reference count is decremented. When the reference count reaches zero, the object is reclaimed. This approach suffers from the inability to reclaim cyclic structures. Each structure is purely self-referential that represents memory space that will not be reclaimed during the program's execution. Because of this limitation, reference counting is generally accompanied by a back-up tracing collector [14] or a complex algorithm to break up and detect the cyclic structures [3, 8].

6.2 Related Work

There are several research efforts recognizing the effect of garbage collection on the throughput. Work by [7] attempted to improve the overall throughput of Java applications by using reference counting for the mature generation space. Work by [11, 25] also attempted to improve the throughput by creating a garbage collection technique that reduces paging, but their techniques need operating system's support. Recent work by [26] proposed a program-level memory management scheme, which tracks the memory usage and number of page faults at program's phase boundaries and adaptively finds a good heap size. But their scheme needs to instrument the application program. Work by [5] compared different GC techniques to investigate the cost of different GC techniques on different heap size and architectures. They did not study the influence of GC techniques on throughput degradation of application servers.

Work by [12] investigated the degradation behavior of Web application servers running on different operating systems including Linux, Solaris 9, FreeBSD, and Windows 2003 servers. They found that Solaris 9 has the most graceful degradation behavior. They also identified the factor that has the greatest effect on the degradation behavior as thread synchronization (waiting time to acquire locks). They reported that Linux threads issue a larger number of system calls during the operation and the thread scheduling policy is inappropriate.

7. CONCLUSIONS

This paper explored the throughput degradation behavior of Java application servers. We found the throughout of Java application servers degrade ungracefully. During heavy workload, a 22% increase in the workload can degrade the throughout by as much as 75%. This result motivated us to investigate what are the major factors affecting the throughput degradation and how do they affect the degradation.

We monitored execution time of three major components in the Virtual Machine: runtime compilation, garbage collection, and synchronization. Our results showed that garbage collection is the major factor because its overhead increases drastically (as much as 50% of the overall execution time) at the heaviest workload. Further studies led us to the following conclusions. First, the assumption that most objects die young is no longer true in application servers. Thus, the Generation Mark-Sweep technique used in the HotSpot VM does not perform well. Second, garbage collection techniques to increase paralellism while greatly improve the maximum throughput, make degradation behavior worse. Third, Ulterior Reference Counting, an incremental generational technique, can positively impacted the degradation behavior. Last, more conservative heap sizing policy only has slightly positive effect on the degradation behavior. However, it can significantly reduce the heap usage (20%) and the number of page fault (10%).

8. ACKNOWLEDGMENTS

This work is support in part by NSF under grant number CNS-0411043. We also thank the reviewers for valuable comments.

9. REFERENCES

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[2] E. Armstrong. Hotspot a new breed of virtual machine. *JavaWorld*, 1998.

[3] D. F. Bacon, C. R. Attanasio, H. Lee, V. T. Rajan, and S. Smith. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–103, 2001.

[4] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. *Lecture Notes in Computer Science*, 2072:207–235, 2001.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2004. ACM Press.

[6] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. . B. Moss. Beltway: Getting Around Garbage Collection Gridlock. In *Proceedings of the Programming Languages Design and Implementatio n*, 2002.

[7] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 344–358, New York, NY, USA, 2003. ACM Press.

[8] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June, 1984.

[9] A. Gupta and M. Doyle. Turbo-charging Java HotSpot Virtual Machine, v1.4.x to Improve the Performance and Scalability of Application Servers. http://java.sun.com/developer/technicalArticles/Programming/turbo/.

[10] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Error-free garbage collection traces: how to cheat and not get caught. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 140–151, New York, NY, USA, 2002. ACM Press.

[11] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM Press.

[12] H. Hibino, K. Kourai, and S. Shiba. Difference of Degradation Schemes among Operating Systems: Experimental Analysis for Web Application Servers. In *Workshop on Dependable Software, Tools and Methods*, Yokohama, Japan, July 2005.

[13] JBoss. Jboss Application Server. Product Literature, Last Retrieved: June 2005. http://www.jboss.org/products/jbossas.

[14] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

[15] J. L. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4):184–195, 1960.

[16] D. Sholler. .NET seen gaining steam in dev projects. In *ZD Net*, http://techupdate.zdnet.com, April 2002.

[17] W. Srisa-an, C. Lo, and J. Chang. Do generational schemes improve the garbage collection efficiency. In *International Symposium on Performance Analysis of Systems and Software (ISPASS 2000*, pages 58–63, Austin, TX, April 24-25, 2000.

[18] W. Srisa-an, M. Oey, and S. Elbaum. Garbage collection in the presence of remote objects: An empirical study. In *International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, 2005.

[19] Standard Performance Evaluation Corporation. SPECjAppServer2004 User's Guide. On-Line User's Guide, 2004. http://www.spec.org/osg/jAppServer2004/docs/UserGuide.html.

[20] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly and Associates, 2003.

[21] D. Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[22] J. Weizenbaum. Symmetric List Processor. *Communications of the ACM*, 6(9):524–544, 1963.

[23] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[24] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, 1992.

[25] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic Heap Sizing: Taking Real Memory into Account. In *Proceedings of the International Symposium on Memory Management*, 2004.

[26] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[27] L. Zhang and C. Krintz. Adaptive code unloading for resource-constrained JVMs. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 155–164, Washington, DC, USA, 2004.

Session C
Mobile and Distributed Systems

Streaming Support for Java RMI in Distributed Environments

Chih-Chieh Yang, Chung-Kai Chen,
Yu-Hao Chang, Kai-Hsin Chung and Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30013, Taiwan
{ccyang, ckchen, yhchang, kschung}@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw

ABSTRACT

In this paper we present novel methodologies for enhancing the streaming capabilities of Java RMI. Our streaming support for Java RMI includes the *pushing* mechanism, which allows servers to push data in a streaming fashion to the client site, and the *aggregation* mechanism, which allows the client site to make a single remote invocation to gather data from multiple servers that keep replicas of data streams and aggregate partial data into a complete data stream. In addition, our system also allows the client site to *forward* local data to other clients . Our framework is implemented by extending the Java RMI stub to allow custom designs for streaming buffers and controls, and by providing a continuous buffer for raw data in the transport layer socket. This enhanced framework allows standard Java RMI services to enjoy streaming capabilities. In addition, we propose aggregation algorithms as scheduling methods in such an environment. Preliminary experiments using our framework demonstrate its promising performance in the provision of streaming services in Java RMI layers.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Experimentation, Languages

Keywords

Java RMI, Streaming Java RMI, Aggregation Scheduling Methods, Novel Applications of Java, Java-based Tools

1. INTRODUCTION

The increasing importance of distributed object-oriented environments for use in parallel and distributed service frameworks has increased interest in efficiently supporting for remote-invocation frameworks, since this layer appears to be a promising paradigm for supporting ubiquitous component communications in heterogeneous network environments. This is also the layer where the well-known software layers such as Java Remote Method Invocation (RMI), .NET Remoting, and CCA (Common Component Architecture) remoting are located.

There has been abundant research in this area recently, which has yielded an open RMI implementation that makes better use of the object-oriented features of Java [1]. The ARMI [2] and Manta [3] systems reduce various drawbacks of RMI and provide new RMI-style systems with extended functionality. KaRMI [4] improves the implementation of RMI by exploiting Myrinet hardware features to reduce latencies. A broad range of RMI applications has also been done in [5]. Our research group has also investigated issues associated with the use of RMI in a wireless environment [6, 7]. The Aroma system [8], which is Java-based middleware, aims to exploit Java RMI to replicate objects so as to ensure both availability and adaptability. The dynamic proxy is used as an interceptor to extend the capabilities of Java RMI. In addition, we have also reported on specifications for RMI programs operating in heterogeneous network environments [9]. In recent years, network streaming becomes a highly popular research topic in computer science due to the fact that a large proportion of network traffic is occupied by multimedia streaming. In the network streaming scenario, data that are retrieved from the network can be processed immediately after a sufficient portion (but not necessarily all) of the total stream has arrived. The streaming technology is applied mostly to multimedia data due to the large sizes of these files, allowing users to listen or view a multimedia file while downloading it. Java RMI allows programmers to rapidly establish distributed object-oriented applications. However, the one-to-one and call-and-wait schemes in Java RMI do not match the properties required for streaming in applications. We therefore aimed to extend Java RMI by inserting extra components so as to implement support for streaming.

In this paper we present our novel methodologies for enhancing the streaming capabilities of Java RMI to provide a flexible and convenient framework for streaming applications. Our streaming support for Java RMI includes the *pushing* mechanism, which allows servers to push data in a streaming fashion to the client site via Java RMI, and the *aggregation* mechanism, which allows the client site in a single remote invocation to gather data from multiple servers that keep replicas of data streams and aggregate partial data into a complete data stream. In addition, our system also allows the client site to *forward* local data for further services. Our framework is implemented by extending the Java RMI stub to allow custom designs for streaming buffers and controls, and by providing a continuous buffer for raw data in the transport layer socket. This enhanced framework allows standard Java RMI services to enjoy streaming capabilities. In addition, we propose aggregation algorithms as scheduling methods in such an environment. Our scheme schedules communications by also considering both the resources and data available at each server. Preliminary experiments using our framework demonstrate its promising performance in the provision of streaming services in Java RMI layers.

The remainder of this paper is organized as follows. Section 2 introduces the basics of Java RMI, and Section 3 introduces the software architecture of streaming RMI. Sections 4, 5, and 6 discuss the three important mechanisms in our framework. Section 7 introduces the user APIs of streaming RMI. The results of our performance evaluation are presented in Section 8, several related works are mentioned in Section 9, and conclusions are drawn in Section 10.

2. THE BASICS OF JAVA RMI

In a distributed system, applications run in different machines in the network. These applications need to communicate with each other to send commands and exchange computation results from time to time. A socket is the basic communication mechanism that is supported in almost every programming language and operating system. Sockets are highly flexible, but they are at a level that is too low for developers because new protocols need to be designed for specific purposes, and they need to encode and decode messages for exchange.

The Java platform provides developers with Java RMI, which utilizes features such as dynamic class loading and reflection of Java, and uses a stub object to manage the method invocations. Developers simply create a stub that is used to make method invocations, which takes care of marshalling the invocation and sending it to the remote RMI server. From the developer's viewpoint, this is just like calling a local object.

The architecture of Java RMI is shown in Figure 1.

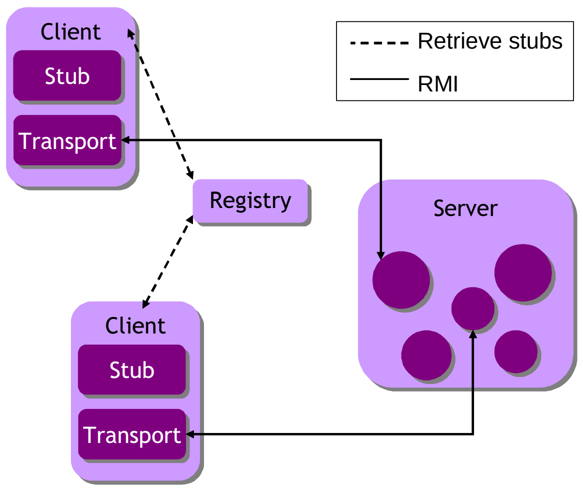


Figure 1: Java RMI architecture.

All RMI servers register with the registry server of their hosting remote objects . RMI clients first look up stubs of a remote object on the registry server, which they subsequently use to call their corresponding remote objects. The stub is very important in Java RMI. It acts as the remote object to the client application by implementing the same interface as the remote object. The client application invokes methods on the stub, and the stub will carry out the methods call on the remote object. It marshals the parameters to the remote JVM, waits for the result, unmarshals the return value, and returns the result to the application. In this work we focus on how to modify the behavior of an RMI stub to integrate streaming ability into Java RMI.

3. SOFTWARE ARCHITECTURE OF STREAMING RMI

The main work of this research was to design an extended architecture for Java RMI to provide capabilities for streaming applications. Figure 2 presents an overview of our proposed architecture, called *streaming RMI*.

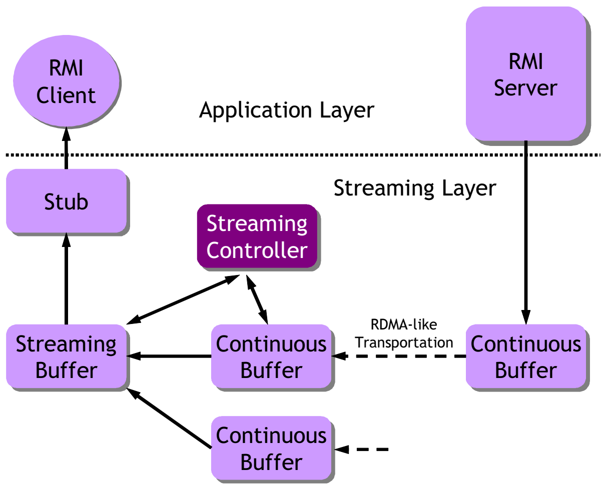


Figure 2: Overview of streaming RMI architecture.

Several important components are needed to support streaming. First, a *continuous buffer* provides the basic buffering of component communications. When a server wants to send data to a client, it writes its own continuous buffer that will push the data through the network to the client. A client reads continuous buffers of its own to receive data from the network. Second, there are two additional components needed in the client-side architecture: *streaming controller* and *streaming buffer*. The streaming controller is responsible for scheduling how servers should send their data, and for aggregating the received data from continuous buffers. The streaming buffer is a repository for aggregated data. With these components, a streaming RMI client can gather data from many different streaming RMI servers simultaneously and aggregate the data received into a new data stream . Figure 3 shows how these new components can be deployed in the layers of standard Java RMI.

Figure 3 extends the Java RMI framework on both the client and server sides. In the client side, we extend the stub to include a streaming controller and a streaming buffer. We also extend the transport layer socket to include continuous buffer support. In the server side, we extend the stub with a loader and a continuous buffer. The detailed behavior of each component is as follows:

Stub The stub of Java RMI is usually generated automatically. In order to add customized features, we disassemble the class file generated by the RMI compiler (`rmic`) to Java source code, and modify its content to intercept method calls. We make it able to create multiple streaming sessions to different servers, and able to pass intercepted method invocations to

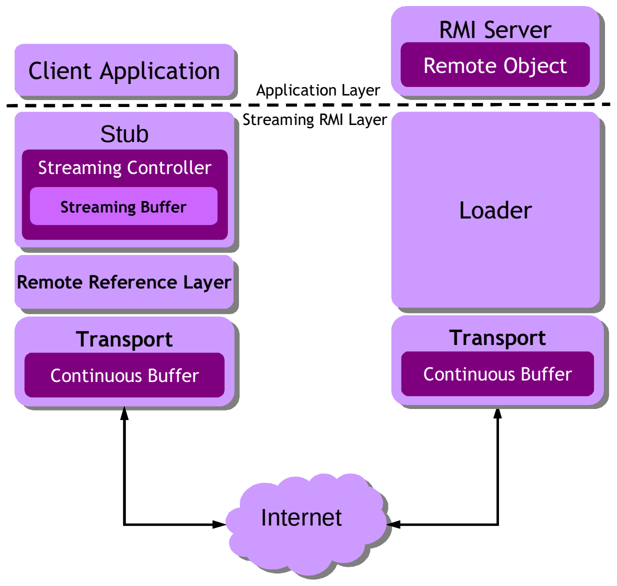


Figure 3: Details of streaming RMI architecture.

the streaming controller that is responsible for parsing their meaning and making necessary modifications before sending the invocations to streaming RMI servers.

Streaming Controller The streaming controller communicates with the stub. It receives remote invocations intercepted by the stub and parses their content. After parsing, the stub may make modifications, distribute the invocations to many streaming RMI servers, or directly return results that already reside in the local streaming buffer. It is also responsible for scheduling which data each server should send to clients, and how this should be achieved. The streaming controller also gathers raw data received in continuous buffers, aggregates them, and stores them in a streaming buffer.

Streaming Buffer The streaming buffer exists only on the client side of a streaming RMI infrastructure, and serves as an aggregation repository. A streaming RMI client may receive raw data from different servers in the network. The streaming controller knows how to aggregate these raw data and store the results in the streaming buffer for applications to retrieve.

Continuous Buffer The continuous buffer is responsible for pushing raw data from the server side to the client side, and so it exists both on the client and server sides and can be treated as a pair. The continuous buffer is in charge of buffering data to send and sending data to the corresponding continuous buffer on the client side at a specific rate. In addition, the upper layer of the client side is able to read data from the local continuous buffer for further processing.

Loader Standard Java RMI on the server side is designed to passively respond to client requests. To maximize efficiency, streaming RMI servers should be able to actively send data to clients. To achieve this, the loader is added to the server-side architecture to periodically load streaming data into the continuous buffer after an initial streaming session is set up. The data in the continuous buffer will then be pushed to clients.

A streaming RMI session is started as follows:

1. The server of the streaming session begins listening for an incoming connection request.
2. A client queries for the specific type of remote object with an RMI registry to obtain a stub.
3. The client application calls a specific initialization method on a stub. The stub will forward the invocation to the streaming controller.
4. The streaming controller parses the method invocation, looks for available streaming RMI servers, creates continuous buffers, and schedules how servers should send streaming data.
5. Initialization messages are sent to streaming RMI servers, as in standard RMI; and continuous buffers create extra connections to server-side continuous buffers to support data pushing (see Section 4).
6. Data are loaded by server-side loaders to continuous buffers and pushed to the client through sockets automatically. The streaming controller collects and aggregates data, which are stored in the streaming buffer after aggregation.
7. If subsequent data requests from client applications can be satisfied by data stored in the streaming buffer, these method invocations will be returned immediately.

4. PUSHING MECHANISM

This section describes pushing, which is the data transmission mechanism in our streaming RMI architecture that replaces the call-and-wait behavior in standard Java RMI. Our automatic mechanism of pushing from a server to clients is achieved by creating a dedicated communication channel for data pushing, with the original channel used by RMI to exchange control messages only. The loader in the server of the streaming RMI infrastructure creates a server socket and listens for incoming connections from clients. In order to serve multiple clients, the loader creates a new thread and activates the corresponding continuous buffer to handle the pushing mechanism of the client each time it accepts a client socket. The client sends the client ID through the socket to the server, allowing the server to identify this client and know where to push the data. After these session setup steps, the continuous buffer of the streaming RMI server starts pushing data to the client.

Application of the pushing mechanism requires a method to coordinate the pushing speed of the server. Because the physical buffer length is always limited, the server may overwrite data not yet received by the client if the data rate is too high, which can cause a fatal system error.

Three flow control policies of the pushing mechanism are considered in our design:

1. Send Only When Consumed: In this scheme the client sends the state of the buffer pointer to the server to determine how much data has been consumed by client applications. The server then pushes this amount of data to the client.
2. Adaptive: In this scheme the servers dynamically adjust the pushing rate according to the buffer status of the client. The adaptation is triggered when the client-side continuous buffer is empty or full for a certain time period. The server will then increase or decrease the pushing speed accordingly.
3. As Fast As Possible: In this scheme the servers push data as fast as possible. This may cause some of the data in client buffers to be overwritten. This policy can be applied when

the data arrival speed is more important than data integrity or when expired data are useless to the user applications.

The application developers can choose the most suitable flow control policy for specific applications, which makes the use of streaming RMI more flexible in various distributed applications.

5. AGGREGATION

When a streaming RMI client requires a specific data stream, there may be multiple streaming servers that are able to supply it. It is likely to be desirable to create multiple connections to these servers for several reasons: (1) this will share the load among the servers (load balancing), (2) the bandwidth from several servers can be used to supply the stream simultaneously so as to ensure smooth playback, and (3) different data streams from different servers can be merged into a new aggregated stream. Information on the availability of streaming servers can be obtained in several ways. For example, one can use a peer-to-peer query (if we choose peer-to-peer environment as our network transport layer) or via a centralized content-locator server. In our current implementation for experiments, we adopt the second approach. A peer-to-peer environment can can be incorporated as well. Information of the streaming servers containing specific content is obtained by the streaming controller of a streaming RMI client simply querying the content locator. After the information of available streaming servers is available, a schedule on how these servers should send the stream needs to be made. The optimal schedule depends on the types of streams we retrieve, examples of which include the same static (fixed-sized) content spread across different servers (such as a movie) and different streams with the same properties (such as video and audio streams of a baseball game) that will be aggregated locally. Our streaming RMI framework makes it possible for developers to customize the scheduling policies that are best for their streams with component interfaces. After scheduling, the schedules are sent out through multiple RMI connections to all of the associated servers.

In this work, we focused on how to gather static data streams from multiple servers that keep replicas of streams and aggregate partial data into a complete data stream. We first introduce some notation used to describe our aggregation algorithm:

- A set of streaming servers: $S = \{s_i | i = 1, \ldots, n\}$.
- A set of data blocks: $D = \{d_j | j = 1, \ldots, m\}$.
- For each streaming server s_i:
  - The supplying bandwidth b_i of s_i.
  - A set of data blocks that exists in s_i: $Blocks(s_i)$.
  - The completeness of data in s_i: $Completeness(s_i)$
  - The amount of content: k_i
- The bandwidth requirement of the streaming session: $Req(d_j)$
- The bandwidth allocation table: $BAT_{m \times n}$

Algorithm 1 presents a mechanism to schedule the same data stream coming from different servers. First, we evaluate the *weight* for each streaming server, which represents their priorities. A server with a higher weight is said to be *preferred*. Three factors are used to evaluate the weight of a server. If the bandwidth of a server is higher, the server is capable of supplying more streaming sessions. Another factor is the completeness of the data stream – when a server has most of a data stream, it is more preferred. It is because

Algorithm 1: Bandwidth allocation for aggregation

```
begin
    /* Evaluate weight of each server    */
    foreach streaming server $s_i$ in $S$ do
        $Weight(s_i) =$
        $\alpha \times \frac{b_i}{Req(d_j)} + \beta \times Completeness(s_i) + \gamma \times (\frac{1}{k_i})$
    end
    /* Sort list $S$ by the weight of each
       server in decreasing order         */
    Sort(List S)
    /* Allocate bandwidth                 */
    for i=1 to n do
        for j=1 to m do
            if $Req(d_j) > 0$ and $d_j$ in $Blocks(s_i)$ then
                $BAT(i, j) = MIN(b_i, Req(d_j))$
                $Req(d_j) = Req(d_j) - BAT(i, j)$
            end
        end
        if every element in $Req$ is 0 then
            Break
        end
    end
end
```

Server ID	Bandwidth	Set of blocks contained	Amount of content
A	10	{0, 1, 2, 3}	10
B	20	{0, 1}	5
C	10	{2, 3}	10

Table 1: A running example for Algorithm 1.

when we allocate supplying bandwidth from a streaming server, the bandwidth will be occupied by the requesting client from the time of admission till the end of the session. If it is only a small portion of the complete data stream, there must be a lot of idle time for the pushing thread of the streaming server, and the allocated bandwidth is wasted. The completeness is by

$$Completeness(s_i) = \frac{size(Blocks(s_i))}{size(D)}, \quad (1)$$

The third factor we considered is the amount of different content on a streaming server. A server is less preferred if it has a larger amount of different data streams, because it is better to first utilize the bandwidth of those with fewer data streams, as they offer less flexibility in the scheduling process. The server that is harder to utilize is given work first whenever it has data that are required. The weight of a server is defined as

$$Weight(s_i) = \alpha \times b_i + \beta \times Completeness(s_i) + \gamma \times \frac{1}{k_i}, \quad (2)$$

where α, β, and γ are coefficients that vary among network environments and parameters.

However, the α in equation (2) which falls in the range from 0 to b_i, might effect on the result weight too much compared with other coefficients. Normalization helps us better choose appropriate coefficients. The normalized weight equation is defined as

$$Weight(s_i) = \alpha \times \frac{b_i}{Req(d_j)} + \beta \times Completeness(s_i) + \gamma \times \frac{1}{k_i}, \quad (3)$$

In Equation (3) we fixed the α coefficient by letting it be divided by the required bandwidth of the data stream. Therefore, we can set the α, β, and γ coefficients from 0 to 1 to calculate the weight.

The input to Algorithm 1 is the set of streaming servers, S. We first evaluate weights for each server, and then sort the list by weight. Next, we use up all of the bandwidth that the topmost server can provide, followed by the second, etc., until all the required bandwidth is satisfied. The experiments described in Section 8 demonstrate the effects of our proposed scheme.

Table 1 lists an example to illustrate how our algorithm works. A stream with four blocks {0, 1, 2, 3} is requested, the bandwidth for each block is 20 and there are three available streaming servers containing blocks. If we choose α=0.1, β=0.5, and γ=1, equation (3) indicates that the weight of server A is $0.05+0.5+0.1 = 0.65$, the weight of server B is $0.1 + 0.25 + 0.2 = 0.55$, and the weight of server C is $0.05 + 0.25 + 0.1 = 0.4$. Server A has the highest priority, so we schedule it first. We allocate all the bandwidth needed by blocks {0, 1, 2, 3} to it. After that, we allocate the bandwidth needed by blocks {0, 1} from server B. If the bandwidth requirement is not satisfied by servers A and B, we allocate bandwidth from server C. If the requirement cannot be satisfied by these three streaming servers, the system reports that the allocation operation has failed.

6. FORWARDING

It is possible that some popular media streams will be downloaded by many requesting clients simultaneously. In this situation, the loading of streaming servers can be reduced if some of the active clients are able to forward their content to other clients. For example, in Figure 4, requesting clients R_2 and R_3 can obtain the data stream from forwarding client R_1.

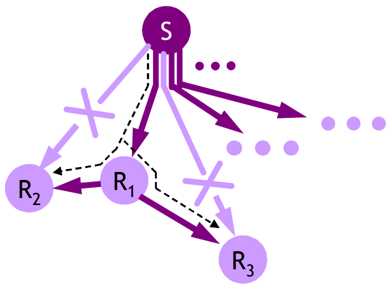


Figure 4: Forwarding behavior.

In our design a requesting client can query the content locator for available forwarding clients, which should reply with several parameters, including the set of data blocks currently in its buffer, the time to live (TTL) for the set of blocks (i.e., the time at which the set will be available), and the bandwidth it can supply. We give the descriptions for those parameters below.

Set of Forwardable Data Blocks The forwarding client may have some blocks buffered before they are consumed by applications. These blocks can be forwarded to other clients, and hence the forwarding client should notify the content locator with the number of blocks available. We assume that the forwarding client is able to forward all subsequent blocks of the stream.

Time To Live We need an estimate of the time at which these blocks will exist in the buffer before they expire; this information is provided to requesting clients.

Supplying Bandwidth The forwarding client also needs to report the amount of bandwidth it can supply.

Algorithm 2: Looking for forwarding clients

```
/* Query the content locator about the
   availability of forwarding clients   */
begin
    foreach response from forwarding clients do
        foreach block in the set of forwardable blocks do
            if the TTL of this block is acceptable then
                Treat this forwarding client as a streaming
                server containing all the required blocks after
                this one
                Break
            end
        end
    end
end
```

Algorithm 2 presents our algorithm for the forwarding case. When a requesting client needs forwarding clients, it sends a query to the content locator as for querying a streaming server. When it receives the responses, it has to check the TTLs of the set of data blocks and dump those that may be expired when the client needs them. If a block passes the test, the requesting client can assume the forwarding client is able to forward all subsequent blocks of the specific stream, and schedule them as resources according to Algorithm 1.

7. SOFTWARE APIS OF STREAMING RMI

The major advantage of the Java RMI programming model is that it hides the details of network communications. Our intention here is to add streaming ability to the model while preserving this advantage. In addition, we provide developers with as much flexibility as possible in manipulating various types of data stream. Developers are able to specify their customized scheduling policies to optimize the performance and aggregation operators for post-processing.

The code shown in Figure 5 illustrates how to write a streaming RMI client, which was actually used in our experiment. (See Section 8.) It simply reads the remote object for specific amount of data and writes the data read into a file. As shown in the example, the initialization steps, such as registry creation and looking up for the stub, are similar to standard RMI. The difference is in the line that `dataObj.initialize()` is invoked, where several streaming session setting parameters (including the address of the content locator, streaming data ID) are wrapped in object `InitSetting` and sent through standard RMI transport channel to participating servers. The parameters are received by the loader at the server for session setup procedures. After these initialization steps, the client simply calls `dataObj.read()` to copy data to its own buffer. The streaming server code is not listed here, because it is almost exactly the same as a standard RMI server program except for the use of the `StreamingData` class as the type of remote object.

We are able to specify customized scheduling policies by specifying the type of scheduler to load in the configuration file (if this is not done, the default scheduler will be applied). The interfaces of the scheduler and operator are listed in Figure 6. A scheduler takes a set of available sources and constraints of the streaming session as input and returns a set of schedule plans once scheduling is completed.

Figure 6 also defines the interface of aggregation operators. Programmers need to implement the `Process()` method, which takes a set of data blocks from different continuous buffers as input and outputs a set of processed data blocks. The use of the two methods `SetNextOperator()` and `GetNextOperator()` allows op-

```
public class StreamingClient {
  public static void main(String args[]) {
    byte[] buf = new byte[100 * 1024]; //read buffer
    int toRead = 10*1024; // read block size = 10kB
    int bytesRead = 0;
    try {
      Registry registry =
        LocateRegistry.getRegistry(host, 9999);
      StreamingInterface dataObj =
        (StreamingInterface)
        registry.lookup("StreamingData");
      String id =
        dataObj.initialize(
        new InitSetting(host, "test.avi")
        );
      File file = new File("copy.avi");
      FileOutputStream os = new FileOutputStream(file);
      while (true) {
        bytesRead = dataObj.read(buf, 0, toRead);
        if (bytesRead == -1) {
          os.close();
          return;
        }
        os.write(buf, 0, bytesRead);
        if (bytesRead == 0)
          Thread.sleep(5); \\ rest for a while
        }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 5: Code listing: initialization of a streaming RMI client.

```
public IScheduler {
  Plan[] Schedule(Node[] sources, Constraints c);
}

public IOperator {
  Block[] Process(Block[] blocks);
  void SetNextOperator(IOperator op);
  IOperator GetNextOperator();
}
```

Figure 6: Code listing: interfaces of scheduler and operator.

erators to be chained to each other to perform a series of operations. For example, we can chain a decompression operator and a merge operator to decompress streams before merging them.

The above examples indicate that creating a streaming RMI application is as simple as creating a standard RMI one. Our streaming RMI framework also provides flexibility for developers to perform various aggregation operations and to apply customized scheduling policies.

8. EXPERIMENTS

We evaluated the performance and overhead of our streaming designs using experiments in which it was deployed in a simple streaming application, and compared it with standard RMI. In addition, we performed a simulation to show the advantages of our proposed aggregation algorithm.

In our first experiment, we evaluated the performance by setting up a streaming server and a requesting client (which is listed in Section 7) over a 100Mbits/sec bandwidth link. The programs are built and tested under JDK version 1.5.0_04 in Microsoft Windows XP machines. The client tried to retrieve a data stream by repeatedly requesting a fixed-sized block. We measured the throughput obtained by varying the block size the client requested in each `read` operation. In streaming RMI, the operation should benefit from our pushing mechanism for its prefetching behavior.

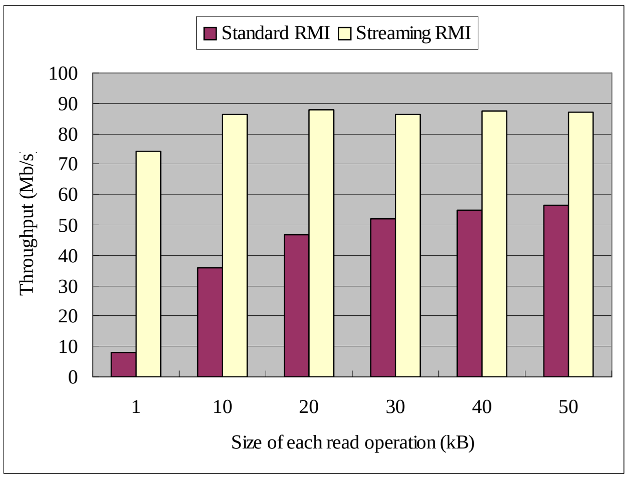


Figure 7: Performance benefits with the pushing mechanism.

Figure 7 presents the performance comparison, and indicates that the smaller the read block size was, the worse standard RMI performed. From the result, we observed that pushing mechanism actually made data streaming effective, because streaming RMI was able to utilize about 85% of network bandwidth provided, while the throughput of standard RMI suffered from frequent remote method invocations to the RMI server. For streaming RMI, method invocations are performed only at initialization or changing streaming settings and data are automatically pushed from server to client without extra remote method invocations, while standard RMI needs client to make a remote method invocation with streaming server for each block requested. Although the situation for standard RMI improved with the increasing size of read blocks, its throughput was still far behind the throughput of streaming RMI.

Read block size (kbyte)	1	10	50	100
Read times	5000	500	100	50
Standard RMI read overhead (byte)	243777	25000	5979	3129
Streaming RMI read overhead (byte)	769	768	769	769
Standard RMI write overhead (byte)	228798	23413	5223	2880
Streaming RMI write overhead (byte)	647	646	647	646

Table 2: The data overhead in sum caused by invocation messages.

In the second experiment, we compared the additional data overhead caused by invocation messages of standard RMI with the overhead in streaming RMI. We set up a streaming server and a client. The client requested for a 5 MB data stream from the server by repeatedly reading a fixed-sized block, and we used a counting socket in RMI transport layer to measure total amount of bytes sent and received from the socket. In standard RMI, each `read` operation is marshalled by its stub and sent to RMI transport layer, but in streaming RMI, the `read` operation performed by a client is handled by *streaming controller* by checking the local continuous buffer for data prefetched from a dedicated pushing channel. As a result, these invocations are handled locally without travelling through the network. We suppose the data overhead from invocation messages is greatly reduced in streaming RMI, and the experimental results proved it.

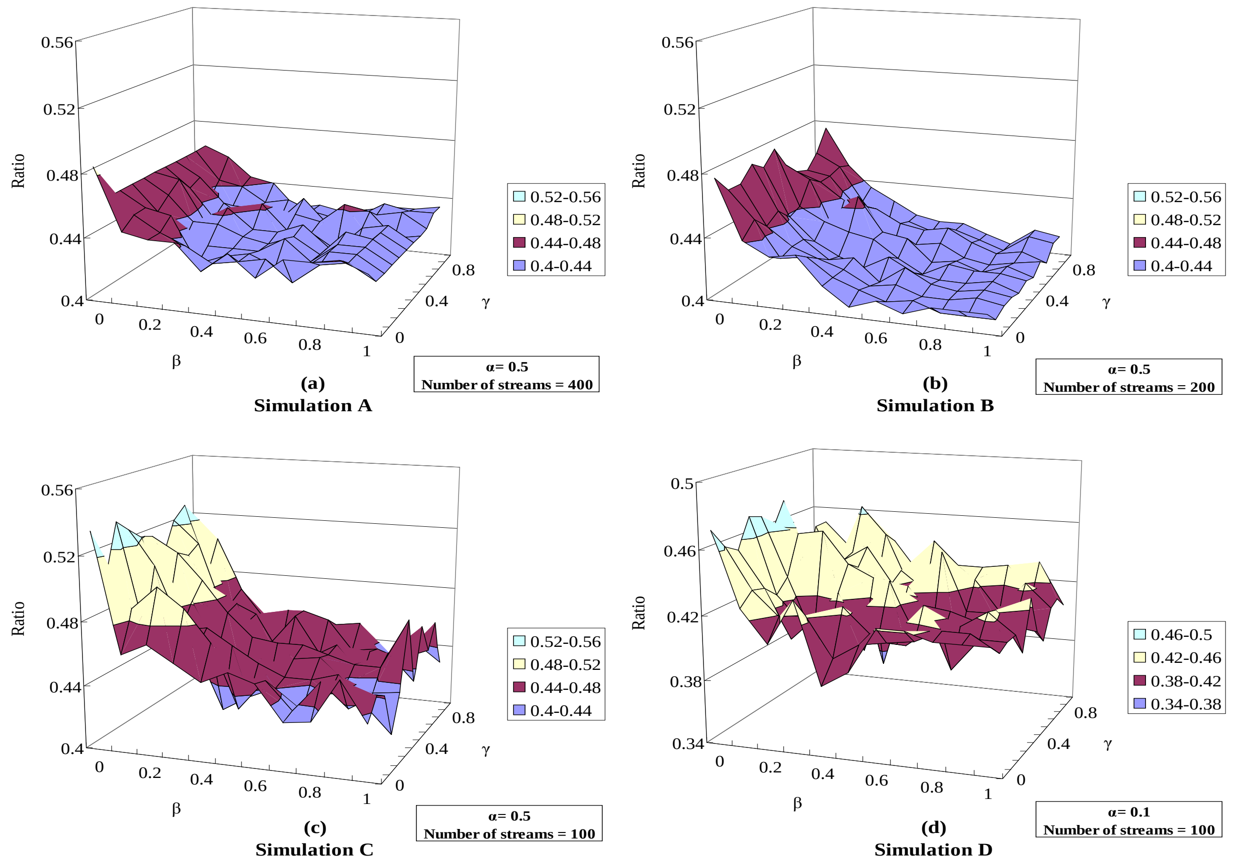


Figure 8: The simulation results with different coefficient weights from our aggregation method.

From Table 2, we are able to see that in standard RMI, the total data overhead was large when the read block size was small. Because the client needed to send more invocation messages for the same stream. With pushing mechanism, the data was prefetched in the local buffer of the client, and the need of exchanging invocation request and response messages was greatly reduced. Thus, no matter which size of the read block was, the total overhead was nearly the same and minimal.

Next, we designed an experimental simulator for investigating the effects of our proposed aggregation policy. We input parameters needed into the simulator, and the simulator runs by the following steps:

1. Creates nodes and bandwidth information.
2. Generates and distributes streaming contents into the nodes.
3. Generates and distributes streaming requests into the nodes.
4. Run the simulation. Scheduling is performed at the time each request is issued.

Table 3 lists the parameters and their values used in the simulation. The number of nodes, the number of different streams, the simulation period, and the size of the data stream were fixed. The number of duplicates of a stream, the bandwidth settings of nodes, and the bandwidth requirements of streams were selected considering behaviors of real life applications, with the samples being uniformly distributed between the upper and lower bounds. We tested the performance under different coefficient sets with a fixed requests. In our simulation, parameters were set as shown in Table 3. The results are presented in Figure 8.

Parameter	**Value**
Number of nodes	1000
Size of data streams (MB)	2
Simulation period (s)	3600
Number of duplicates	1, 2, 3, . . . , 100
In-bound bandwidth (kB/s)	8, 16, 32, 64, 128
Out-bound bandwidth (kB/s)	8, 16, 32
Bandwidth requirement (kB/s)	8, 16, 32, 64, 128, 256, 512
Number of requests	5000
β, γ	0, 0.1, 0.2, 0.3,. . . , 1.0

Simulation	(a)	(b)	(c)	(d)
α	0.5	0.5	0.5	0.1
Number of different streams	400	200	100	100

Table 3: Parameter sets used in the simulation.

The x-axis is β, the y-axis is γ, and the z-axis represents *the ratio of average waiting time* of systems deploying our aggregation policy compared with those without aggregation policy (which means that only nodes with complete data streams are able to supply the request). Thus the the lower z values represents the better results.

In Figures 8(a), 8(b), and 8(c), there is a clear trend that when the coefficient β (which represents completeness of a data stream) was increased, the ratio of average waiting time is lower. This shows the importance of β in our aggregation policy. Also by observing the three figures, we know that when the number of different streams reduced, the slope was getting steeper but the results are not overall worse. It indicates the possiblity for a system manager to adjust the coefficients to get better performance, or the system automatically adjusts them from machine learning experiences when stream sources become fewer.

And from Figures 8(c) and 8(d), we found that when α (which represents the importance of available bandwidth) was lower, the effect of β and γ is magnified, and achieved overall better results. We suppose that with aggregation, the bandwidth requirement of a streaming session is easy to be satisfied while there are lots of duplicates in the network. Thus the importance of α is reduced.

The simulation results indicate that our aggregation policy greatly reduces the average waiting time, and that better performance is obtained in all cases when all three factors are considered. It also demonstrates that the completeness of a stream is the most important factor, and suggests that tuning of coefficients may further improve system performances.

9. RELATED WORK AND DISCUSSION

Streaming multimedia has great increased in popularity in recent years, in applications such as video on demand. A streaming module implements "play while downloading" instead of "play after downloading". Several techniques have been proposed to implement streaming on the Internet. A distributed video streaming is presented in [12], where each session involves multiple replicated video servers. The idea is similar to our aggregation mechanism in Section 5. However, the system lacks scalability without mechanisms like forwarding, so the system may be overloaded if the number of clients grows. In [13], multiple reliable servers are installed across the network to act as a software router with multicast capability. This approach can allow a client to obtain data not only from the source but also from the software router, thus alleviating the bandwidth demand on the source, which is similar to the effect of the forwarding mechanism.

The Object Management Group has defined a specification for the control and management of audiovisual streams [14], based on the CORBA (Common Object Request Broker Architecture) reference model [15]. This specification defines the mode of implementation of an open distributed multimedia streaming framework. In [16], TAO audiovisual streaming services are addressed and provide applications with the advantages of using CORBA IIOP in their stream establishment and control module, while allowing the use of more efficient transport-layer protocols for data streaming. An explicit open bindings mechanism is proposed in [17], which allows the programmer to set up an additional transport connection between CORBA objects. This connection can be used for more precise control over streaming session. This is similar to our work in extending a remote method invocation mechanism to provide streaming ability. Nevertheless, it only focuses on the data delivery process, and lacks data aggregation mechanism for users to better manipulate the content of streaming data.

Compared with previous research works, our work is designed to extend the streaming capabilities for standard Java RMI, which is a well known language feature of a popular object-oriented programming language. We also provide software APIs for programmers to build streaming applications on top of our framework.

In this paper we provide an alternative method on handling streaming data. We model data streams into objects which can be accessed remotely by our extended RMI mechanism. Our design also presents a framework allowing the data stream objects to distribute the delivering jobs to other duplicated source and the client to schedule and aggregate the data. Different scheduling algorithms can be plugged in this framework to provide adequate functionality. We believe this extended RMI can be used as prime functions exposed to the users of Data Stream Management Systems (DSMS) such as Aurora [18] and Nile [19]. In JSR 158: Java Stream Assembly [20], a model for stream manipulation is proposed, which is similar with what we would like to achieve in aggregation mechanism. We might consider the possibilities of integrating these techniques into our framework in the future.

As to the issue of garbage collection, the processing and delivering of streaming data might be interfered when garbage collection occurs. In our implementation we reuse most objects such as temporary buffers to reduce the need of heap allocation. Nevertheless this can not completely avoid garbage collection. To guarantee smooth streaming processing, a system incorporating performance issues and models of real-time garbage collection schemes is also be needed.

10. CONCLUSIONS

In this paper we present our novel methodologies for enhancing the streaming capabilities of Java RMI by inserting additional components such as streaming controller, streaming buffer, continuous buffer, and loader. These components are designed for supporting core streaming technologies with pushing, aggregation, and forwarding mechanisms. Our work also comes with a software design that is ready for use. Preliminary experiments performed using our framework demonstrate the promising performance of our proposed schemes in providing streaming services in Java RMI layers.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable suggestions and comments that have helped to improve the presentation and quality of our paper.

This work was supported in part by Ministry of Economic Affairs under grant no. 95-EC-17-A-01-S1-034, by National Science Council under grant no. NSC 94-2220-E-007-019, NSC 94-2220-E-007-020, NSC 94-2213-E-007-074, NSC 95-2752-E-007-004-PAE, and NSC 95-3113-P-007-002, and by Institute for Information Industry under grant no. 95-EC-17-A-99-R1-0461.

11. REFERENCES

[1] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.

[2] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, 1997.

[3] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java remote method invocation. In: *The Proceedings of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Atlanta, GA, pp. 173–182, May 1999.

[4] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In: *Proceedings of the ACM Java Grande Conference*, San Francisco, CA, pp. 152–157, Jun. 1999.

[5] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, 1998.

[6] P. C. Wey, J. S. Chen, C.-W. Chen, and J.-K. Lee. Support and optimization of Java RMI over Bluetooth environments. In: *Proceedings of the ACM Java Grande - ISCOPE Conference*, Seattle, WA, 17:967–989, Nov. 2002.

[7] C.-W. Chen, C.-K. Chen, J.-C. Chen, C.-T. Ko, J.-K. Lee, H.-W. Lin, and W.-J. Wu. Efficient support of Java RMI over heterogeneous wireless networks. In: *Proceedings of ICC*, Paris, France, pp. 1391–1395, Jun. 2004.

[8] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Interception in the Aroma system. In: *Proceedings of the ACM Java Grande Conference*, San Francisco, CA, pp. 107–115, Jun. 2000.

[9] C.-K. Chen, C.-W. Chen, and J.-K. Lee. Specification and architecture supports for component adaptations on distributed environments. In: *Proceedings of IPDPS*, Santa Fe, NM, pp. 47a, Apr. 2004.

[10] C.-K. Chen, Y.-H. Chang, C.-W. Chen, Y.-T. Chen, C.-C. Yang, and J.-K. Lee. Efficient switching supports of distributed .NET Remoting with network processors. In: *Proceedings of ICPP*, Oslo, Norway, pp. 350–357, Jun. 2005.

[11] National Science Council (NSC). *Research Excellence Project*. http://www.ccrc.nthu.edu.tw/PPAEUII/.

[12] T. Nguyen and A. Zakhor. Distributed video streaming over Internet. In: *Proceedings of SPIE/ACM MMCN*, San Jones, CA, Jan. 2002.

[13] J. Jannotti, D. K. Gifford, and K. L. Johnson. Overcast: reliable multicasting with an overlay network. In: *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, pp. 209–301, Oct. 2000.

[14] Object Management Group. Control and management of A/V streams specification. OMG document telecom, 97-05-07 edn., Oct. 1997.

[15] Object Management Group. The common object request broker: architecture and specification, edn. 2.2, Feb. 1998.

[16] S. Mungee, N. Surendran, D. C. Schmidt. The design and performance of a CORBA audio/video streaming service. In: *Proceedings of the Hawaii International Conference on System Sciences*, Maui, Hawaii, pp. 8043, Jan. 1999.

[17] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In: *Proceedings of ICCDS*, Annapolis, Maryland, pp. 128, May 1998.

[18] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management In: *The VLDB Journal*, 12(2):120–139, 2003.

[19] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. In: *Proceedings of ICDE*, Boston, MA, pp. 851-, Mar 2004.

[20] Sun Microsystems, Inc. JSR 158: Java Stream Assembly. http://jcp.org/en/jsr/detail?id=158.

Enabling Java Mobile Computing on the IBM Jikes Research Virtual Machine

Giacomo Cabri
Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
cabri.giacomo@unimore.it

Letizia Leonardi
Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
leonardi.letizia@unimore.it

Raffaele Quitadamo
Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
quitadamo.raffaele@unimore.it

ABSTRACT

Today's complex applications must face the distribution of data and code among different network nodes. Java is a wide-spread language that allows developers to build complex software, even distributed, but it cannot handle the migration of computations (i.e. threads), due to intrinsic limitations of many traditional JVMs. After analyzing the approaches in literature, this paper presents our research work on the IBM Jikes Research Virtual Machine: exploiting some of its innovative VM techniques, we implemented an extension of its scheduler that allows applications to easily capture the state of a running thread and makes it possible to restore it elsewhere (i.e. on a different hardware or software architecture, but still with a version of JikesRVM installed). Our thread serialization mechanism provides support for both proactive and reactive migration of single- and multi-threaded Java applications. With respect to previous approaches, we implemented the mobility framework without recompiling a previous JVM source code, but simply extending its functionalities with a full Java package.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features D.1.3 [**Concurrent Programming**]: Distributed Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms

Design, Experimentation, Languages.

Keywords

Java Virtual Machine, thread persistence, code mobility, distributed applications.

1. INTRODUCTION

Modern distributed systems are becoming more and more complex, leading to the need for flexibility, that has to be considered very desirable, if not mandatory, when large scale distributed computations are performed. Conventional software components, scattered among network nodes, provide services to other components or to end users, but are often statically bound to their hosting environment. This view is being challenged by technical developments that introduce a degree of mobility in distributed systems. Wireless LANs and mobile devices have already highlighted the potentials of physical mobility [15, 4]. Code mobility is instead reshaping the logical structure of modern distributed systems as it enriches software components (in particular, execution units) with the capability to dynamically reconfigure their bindings with the underlying execution environments. The concept is simple and elegant; an object (that may be active or passive) that resides on one node is migrated to another node where execution is continued. The main advantages of mobile computations, be they object-based or not, are as follows: *(i) load balancing*: distributing computations (e.g. Java threads) among many processors as opposed to one processor gives faster performance for tasks that can be fragmented; (ii) *communication performance*: active objects that interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction; (iii) *availability*: objects can be moved to different nodes to improve the service and provide better failure coverage or to mitigate against lost or broken connections; *(iv) reconfiguration*: migrating objects permits continued service during upgrade or node failure. (v) *location independence*: an object visiting a node can rebind to generic services without needing to specifically locate them. They can also move to take advantage of services or capabilities of particular nodes.

It can be argued that Java has all the prerogatives to make thread mobility possible: its platform-independent bytecode language and the support for object serialization. Regular Java object can be easily made persistent or migrated to other machines, by means of the JVM built-in serialization facility. Java threads are coherently presented to the programmer as objects as well, but their serialization does not produce the desired effect of "capturing their current execution flow and resuming it elsewhere"; it is just the `java.lang.Thread` object, with its fields, that is serialized, while the real execution flow is still tightly bound to the execution environment. The Java programming language does not therefore support the migration of execution units (e.g. threads), which exists for other languages and specific operating systems [17].

Some kind of framework or "JVM enhancement" is thus needed to enable mobile computations in distributed Java applications.

Several approaches have been proposed and experimented in order to add thread migration capability to the Java run-time environment. The bulk of them provide only a "weaker form" of mobility, where the system simply allows the migration of the code and some data, but discards the execution state (e.g. the Java method stack, context registers and instruction pointer). In Section 2 of this paper, we argue that *weak mobility* is not always the best choice (e.g. particularly when dealing with complex parallel computations) and is sometimes definitely unsuitable in some cases (e.g. if the program has a recursive behavior). Therefore, after having shortly discussing the main issues of thread migration in the literature and its motivations with some potential applications, in Section 3 we outline the main contributions of this paper: a novel approach towards the provision of Java thread strong migration on top of the IBM Jikes Research Virtual Machine, exploiting some well-established object-oriented language techniques (e.g. On-Stack Replacement, type-accurate garbage collectors, quasi-preemptive Java thread scheduler, etc.) to capture the execution state of a running thread; an extension of the IBM JikesRVM scheduler that Java programmers can dynamically enable, simply importing a Java package into their mobile Java applications. The choice of JikesRVM is strongly motivated by the fact that it was born as a VM specifically targeted to multiprocessor SMP servers. Likewise, our framework focuses on these kinds of hardware, where strong mobility is mostly required. In Section 4, we present our performance tests, made writing a benchmark based on the Fibonacci recursive algorithm, and in Section 5 we prospect our future research work on this mobility framework. Conclusions are drawn in Section 6.

2. BACKGROUND AND MOTIVATIONS

This section introduces the main issues to address when designing a thread migration mechanism and provides a brief overview of proposed approaches in literature. It also sketches some real applications that would benefit from the work explained later.

2.1 Motivations

The choice of thread mobility, when designing distributed Java applications, has to be carefully motivated, since it is not always the best one in most simple cases. Distributed and parallel computations can be considered perhaps the "killer application" of such technique. For instance, complex elaborations, possibly with a high degree of parallelism, carried out on a cluster of servers would certainly benefit from a thread migration facility in the JVM. Well-know cases of such applications are mathematical computations, which are often recursive by their own nature (e.g. fractal calculations) and can be parallelized to achieve better elaboration times.

Another field of application for mobile threads is load balancing in distributed systems (e.g. in the Grid Computing field), where a number of worker nodes have several tasks appointed to them. In order to avoid overloading some nodes while leaving some others idle (for a better exploitation of the available resources and an increased throughput), these systems need to constantly monitor the execution of their tasks and possibly re-assign them, according to an established load-balancing algorithm. As we will see later, a particular kind of thread migration (called *reactive migration*), that we provide in our framework, fits very well the requirements of these systems.

2.2 Thread mobility issues

Java threads are often considered a valid example of a so-called execution unit [11], performing their tasks in a computational environment (i.e. the JVM), but without any possibility of detaching from their native environment. An execution unit is conventionally split into three separate parts that are supposed to be movable to achieve the overall mobility of the execution unit:

- the *code segment* (i.e. the set of compiled methods of the application);
- the *data space*, a collection of all the resources accessed by the execution unit. In an object-oriented system, these resources are represented by objects in the heap;
- an *execution state*, containing private data as well as control information, such as the call stack and the instruction pointer.

Weakly mobile threads can transfer their execution, bringing only code and some data, while the call stack is lost. From the architectural standpoint, it is relatively easy to implement *weak mobility* on top of the JVM, because the Java language provides very powerful tools for that purpose: *object serialization* is used to migrate data, such as objects referenced by the program; *bytecode and dynamic class-loaders* facilitate the task of moving the code across distant JVMs, hosted by heterogeneous hardware platforms and operating systems.

```
public class MyAgent extends Agent {

protected boolean migrated = false;
// indicates if the agent has moved yet

public void run(){
   if( ! migrated ){
      //…things to do before the migration
      migrated = true;
      try{ migrate(newURL("nexthost.unimore.it");
      }catch(Exception e){ migrated = false; }
   }
   else{ // things to do on the destination host
      // possibly other if/else to handle other
      // conditions…
      }
}}
```

Figure 1. An example of a weak mobile agent.

From an application point of view, weakly mobile systems usually force the programmer to write code in a less natural style: extra programming effort is required in order to manually save the execution state, with flags and other artificial expedients. For instance, Mobile Agents (MA) [9] are usually weakly mobile execution units, used in many scenarios: e.g. distributed information retrieval, online auctions and other systems where they have to follow the user's movements, for migrating from and to the user's portable device (mobile phone, PDA, etc.). A simple weak agent is shown in Figure 1. The point is that, with weak mobility, it is as the code routinely performs rollbacks. In fact, looking at the code in Figure 1, it is clear how, after a successful `migrate()` method call that causes the agent migration, the code does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the `run()` method (on the destination machine, of course), and thus there is a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.

A *strongly* mobile thread has the ability to migrate its code and execution state, including the program counter, saved processor registers, return addresses and local variables. The component is suspended, marshaled, transmitted, unmarshaled and then restarted at the destination node without loss of data or execution state. *Strong mobility* turns out to be far more powerful where complex distributed computations are required, as it preserves the traditional programming style of threads, without requiring any code rollback or other expedients: it reduces the migration programming effort to the invocation of a single operation (e.g. a `migrate()` method) and leads to cleaner implementations. Despite these advantages, many systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state. Moreover, a weakly mobile system gives the programmer more control over the amount of state that has to be transferred, while an agent using strong migration may bring unnecessary state, increasing the size of the serialized data.

2.3 Related work

Several approaches have been proposed so far to overcome the limitations of the JVM as concerns the execution state management. The main decision that each approach has to take into account is how to capture the internal state of threads, providing a fair trade-off between performances and portability. In literature, we can typically find two categories of approaches:

- modifying or extending the source code of existing JVMs to introduce APIs for enabling migration (*JVM-level approach*);
- translating somehow the application's source code in order to trace constantly the state of each thread and using the gathered information to rebuild the state remotely (*application-level approach*).

2.3.1 JVM-level approach

The former approach is, with no doubt, more intuitive because it provides the user with an advanced version of the JVM, which can completely externalize the state of Java threads (for thread serialization) and can, furthermore, initialize a thread with a particular state (for thread de-serialization). The kind of manipulations made upon the JVM can be several.

The first proposed projects following the JVM-level approach like Sumatra [22], Merpati [21], JavaThread [6] and NOMADS [23], extend the Java interpreter to precisely monitor the execution state evolution. They, usually, face the problem of stack references collection modifying the interpreter in such a way that each time a bytecode instruction pushes a value on the stack, the type of this value is determined and stored "somewhere" (e.g., in a parallel stack). The drawback of this solution is that it introduces a significant performance overhead on thread execution, since additional computation has to be performed in parallel with bytecode interpretation. Other projects tried to reduce this penalization avoiding interpreter extension, but rather using JIT (Just In Time) re-compilation (such as Jessica2 [28]) or performing type inference only at serialization time (and not during thread normal execution). In ITS [5], the bytecode of each method in the call stack is analyzed with one pass at serialization time: the type of stacked data is retrieved and used to build a portable data structure representing the state. The main drawback of every JVM-level solution is that they implement special modified JVM versions that users have often to download; therefore they are forced to run their applications on a prototypal and possibly unreliable JVM.

2.3.2 Application-level approach

In order to address the issue of non-portability on multiple Java environments, some projects propose a solution at the application level. In these approaches, the application code is filtered by a pre-processor, prior to execution, and new statements are inserted, with the purpose of managing state capturing and restoration. Some of these solutions rely on a bytecode pre-processor (e.g. JavaGoX [18] or Brakes [25]), while others provide source code translation (e.g. Wasp [12], JavaGo [19], Wang's proposal [26]). Two of them [19, 26] hide a weak mobility system behind the appearance of a strong mobility one: they, in fact, re-organize "strongly-mobile" written code into a "weakly-mobile" style, so that weak mobility can be used instead. Portability is achieved at the price of a slowdown, due to the many added statements.

2.3.3 Discussion

Starting from the above considerations, we have decided to design and implement a strong thread migration system able to overcome many of the problems of the above-explained approaches. In particular, our framework is written entirely in Java and it does neither suffer performance overheads, due to bytecode instrumentations, nor reliability problems, because the user does not have to download a new, possibly untrustworthy, version of JikesRVM. The framework is capable of dynamically installing itself on several recent versions of JikesRVM (we carried out successful tests starting from release 2.3.2). In fact, every single component of the migration system has been designed and developed to be used as a normal Java library, without requiring rebuilding or changing the VM source code. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned JVM-level and Application-level approaches. Other midway approaches [14] exploit the JPDA (Java Platform Debugger Architecture) that allows debuggers to access and modify runtime information of running Java applications. The JPDA can be used to capture and restore the state of a running program, obtaining a transparent migration of mobile agents in Java, although it suffers from some performance degradation due to the debugger intrusion.

3. ENABLING THREAD MIGRATION ON TOP OF JIKESRVM

In this section we will describe how we implemented our strong migration mechanism on top of the IBM Jikes Research Virtual Machine (RVM). The JikesRVM project was born in 1997 at the IBM T.J. Watson Laboratories [1] and it has been recently donated by IBM to the open-source community. Two main design goals drove the development of such successful research project [2]: (i) supporting high performance Java servers; (ii) providing a flexible research platform "where novel VM ideas can be explored, tested and evaluated".

In this research virtual machine, several modern programming language techniques have been experimented and, throughout this presentation, we will focus mainly on those features that are most strategic to our system. The proposed description will follow the migration process in its fundamental steps, from thread state capturing to resuming at the destination machine.

3.1 Our JikesRVM extension

When the programmer wants to endow her threads with the capability to migrate or be made persistent on disk, the first simple thing to do is to import the `mobility` package, which exposes the `MobileThread` class. The latter inherits directly from the `java.lang.Thread` class and has to be sub-classed by user-defined threads.

The configuration of the scheduler, when our JikesRVM extension is installed, is reported in Figure 2: the idea is that now, with mobility, threads can enter and exit the local environment through some migration channels. These channels are represented in Figure 2 using the classical notation of *queuing networks*. The software components added by our infrastructure are highlighted with boxes and it can be clearly seen how these parts are dynamically integrated into JikesRVM scheduler, when the programmer enables the migration services.

The single *output channel* is implemented through a regular thread, which runs in a loop, performing the following actions:

1) extract a mobile thread from the migration queue, where these threads wait to be transferred;
2) establish a TCP socket connection with the destination specified by the extracted thread;
3) if the previous step is successful, then capture the complete thread state data (i.e. the `MobileThread` object and the sequence of frames into its call stack, as better explained in Subsection 3.3);
4) serialize those data into the socket stream (`java.io.ObjectOutputStream`);
5) close the connection with the other end-point.

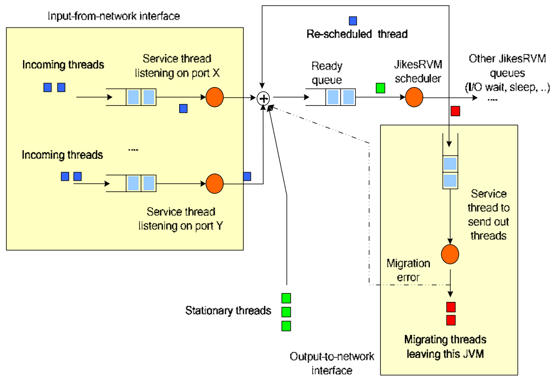

Input-from-network interface
Incoming threads
Service thread listening on port X
Incoming threads
....
Service thread listening on port Y
Re-scheduled thread
Ready queue
JikesRVM scheduler
Other JikesRVM queues (I/O wait, sleep, ..)
....
Service thread to send out threads
Migration error
Stationary threads
Output-to-network interface
Migrating threads leaving this JVM

Figure 2. Adding migration channels to JikesRVM scheduler

One or more *input channels* are implemented by means of regular threads listening on specific TCP ports. When a connection request is issued from the network, they simply do the followings:

1) open the connection with the requesting host;
2) read thread state data from the socket stream (`java.io.ObjectInputStream`);
3) re-establish a local instance of the arrived thread and resume it;
4) close the connection socket.

A first observation is that non-mobile ("stationary" as in Figure 2) threads in the system are not influenced at all by the infrastructure built upon the scheduler. Only those threads that inherit from our enhanced `MobileThread` class are interested in the added migration services.

3.2 Proactive migration vs. reactive migration

There are two ways for a `MobileThread` to get queued into the migration queue, waiting for the output channel to transfer it [11]:

- the mobile thread autonomously determines the time and destination for its migration, calling the `MobileThread.migrate(URL destination)` method (*proactive migration*);
- its movement is triggered by a different thread that can have some kind of relationship with the thread to be migrated, e.g. acting as a manager of roaming threads (*reactive migration*).

Exploiting JikesRVM features, we successfully implemented both migration types, in particular the *reactive migration*. As anticipated in Subsection 2.1, an application, in which reactive migration can be essential, is a load-balancing facility in a distributed system. If the virtual machine provides such functionality to authorized threads, a load monitor thread may want to suspend the execution of a worker thread A, assign it to the least overloaded machine and resume its execution from the next instruction in A's code. This form of transparent externally-requested migration is harder to implement with respect to the proactive case, mainly because of its asynchronous nature.

Proactive migration raises, in fact, less semantic issues than the reactive one, though identical to the latter from the technological/implementation point of view: in both cases we have to walk back the call stack of the thread, extract the meaningful frames and send the entire thread data to destination (see the following two subsections for more details). The fundamental difference is that proactive migration is synchronized by its own nature (the thread invokes `migrate()` when it means to migrate), while for reactive migration the time when the thread has to be interrupted could be unpredictable (the requester thread notifies the migration request to the destination thread, but the operation is not supposed to be instantaneous). Therefore, in the latter case, the critical design-level decision is about the degree of asynchronism to provide. In a few words, the question is: should the designated thread be interruptible anywhere in its code or just in specific safe migration points?

We chose to provide a more coarse-grained migration in the reactive case. Our choice has a twofold motivation: (i) designing the migration facility is simpler; (ii) decreasing migration granularity reduces inconsistency risks. Although these motivations can be considered general rules-of-thumb, they are indeed related to the VM we adopted. In fact, the scheduling of the threads in JikesRVM has been defined as *quasi-preemptive* [1], since it is driven by JikesRVM compilers. In JikesRVM, Java threads are objects that can be executed and scheduled by several kernel-level threads, called *virtual processors*, each one running on a physical processor. What happens is that the compiler introduces, within each compiled method body, special code (*yieldpoints*) that causes the thread to request its virtual processor if it can continue the execution or not. If the virtual processor grants the execution, the virtual thread continues until a new yieldpoint is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread. In particular, when the thread reaches a certain yieldpoint (e.g. because its time slice is expired), it prepares itself to dismiss the scheduler and let a context switch occur. The invoked function to deal with a reached yieldpoint is the static method `VM_Thread.yieldpoint()`. If we allow a reactive migration with a too fine granularity (i.e. potentially at any yieldpoint in thread's life), inconsistency problems are guaranteed. The thread can potentially lose control in any methods, from its own user-implemented methods to internal Java library methods (e.g. `System.out.println()`, `Object.wait()` and so forth). It may occur that a critical I/O operation is being carried out and a blind thread migration would result in possible inconsistency errors.

We are currently tackling the reactive migration issues thanks to JikesRVM yieldpoints and the JIT compiler. In order to make mobile threads interruptible with the mentioned coarse granularity, we introduced the *migration point* concept: migration points are always a subset of yieldpoints, because they are reached only if a yieldpoint is taken. The only difference is that migration points are inserted only:

1) in the methods of the `MobileThread` class (by default);
2) in all user-defined class implementing the special `Dispatchable` interface (*class-level granularity*);
3) in those user-methods that are declared to throw `DispatchablePragmaException` (*method-level granularity*).

The introduction of a migration point forces the thread to check also for a possibly pending migration request (notified reactively by another thread through the same `migrate()` method invoked on the target `MobileThread`). If the mobile thread takes the migration point, it suspends its execution locally and waits in the migration queue (described in Subsection 3.1) until the service thread, responsible for the output channel, selects it and starts the necessary migration operations (see the next two subsections). The code for these additional tests is partly reported in Figure 3. This approach has several advantages: firstly, it rids us of the problem of unpredictable interruptions in internal Java library methods (not interested by migration points at all); then, it also gives the programmer more control over the migration, by letting her select those safely interruptible methods; last but not least, it leaves the stack of the suspended thread in a well-defined state, making the state capturing phase simpler. We achieved the insertion of migration points, simply patching *at runtime* a method of the JIT compiler (the source code of the VM is left untouched and one can use every OSR-enabled version of the JikesRVM). As we already mentioned, yieldpoints are inserted by the JIT compiler, when it compiles a method for the first time. These yieldpoints are installed into method prologues, epilogues and loop heads by the `genThreadSwitchTest()` method of the compiler. In order to force the compiler to insert our migration points instead of yieldpoints in the methods listed above, we patched the `genThreadSwitchTest()`method with an internal method of the framework: the new method has a code nearly identical to the old one, except for the special treatment of the three cases listed above. In these cases, the thread enters the code in Figure 3 and is auto-suspended, waiting to be serialized by the output channel described in Subsection 3.1. We must point out that JikesRVM's compiler does not allow unauthorized user's code to access and patch internal runtime structures. User's code, compiled with a standard JDK implementation, will not have any visibility of such low-level JikesRVM-specific details.

```
static void migrationPoint() throws NoInlinePragma
{
   if (migrationRequested){
   // Process a migration point
   outputChannel.queue(Thread.getCurrentThread());
   suspend(); // thread-switch beneficial
   // The thread is resumed here at the
   // destination
   }
   else // Process a regular  yieldpoint
   {
      yieldpoint();
   }
}
```

Figure 3. The method that deals with a migration point

3.3 Capturing the execution state of a thread

When the service thread, owner of the output channel described in Subsection 3.1, selects a `MobileThread` candidate to serialization, it starts a walk back through its call stack, from the last frame to the `run()` method of the thread..

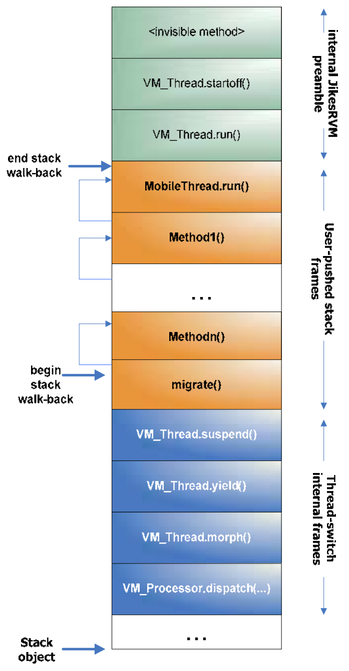


Figure 4. The stack walk-back of a suspended `MobileThread`

This jumping is shown schematically in Figure 4, where the stack is logically partitioned into three areas: *(i) internal preamble frames*, which are always present and do not need to be migrated; (ii) *user-pushed frames*, to be fully captured as explained later; (iii) *thread-switch internal frames*, which can be safely replaced at the destination and, thus, not captured at all.

```
MobileFrame extractSingleFrame() {

  /*Move to the previous user frame (walk-back)*/
   if(!moveToPreviousFrame())
    return null; //no more user frames to capture

  /*Extract the state in the specified object*/
  return defOSRExtractor.extractState(...);

}
```

Figure 5. The `extractSingleFrame()` method of the `FrameExtractor` class

A special utility class, called `FrameExtractor`, has been implemented in our framework, with the precise goal of capturing all the frames in the user area in a portable bytecode-level form. The most interesting method of this class is the `extractSingleFrame()` method reported in the code of Figure 5. This method uses an *OSR extractor* to capture the frame state representation and returns it to the caller, ready to be serialized and sent to destination or to be checkpointed on disk.

3.3.1 The JikesRVM OSR Extractor

The OSR (On-Stack Replacement) extractor is another fundamental component of the framework: it takes inspiration from the OSR extractors provided by JikesRVM [10], though it has been re-written for the purposes of our project.

The OSR technique was introduced in JikesRVM, with a completely different objective: enabling adaptive re-compilation of hot methods. In fact, JikesRVM can rely not only on a baseline compiler but also on an optimized one [7]. Every bytecode method is initially compiled with the baseline compiler, but when the Adaptive Optimization System (AOS) [3] decides that the current executing method is worth being optimized, the thread is drawn from the ready queue and the previous less-optimized frame is replaced by a new more-optimized frame. The thread is then rescheduled and continues its execution in that method. This technique was first pioneered by the Self programming language [8]. An innovative implementation of the OSR was integrated into the JikesRVM [10], which uses *source code specialization* to set up the new stack frame and continue execution at the desired program counter. The transition between different kinds of frames required the definition of the so-called *JVM scope descriptor* that is "the compiler-independent state of a running activation of a method" based on the stack model of the JVM [13]. When an OSR is triggered by JikesRVM, the scope descriptor for the current method is retrieved and is used to construct a method, in bytecode, that sets up the new stack frame and continues execution, preserving semantics.

3.3.2 Our modified OSR Extractor

The JikesRVM OSR frame extractor has been rewritten for the purpose of our mobility framework (we called it `OSR_MobilityExtractor`) to produce a frame representation, suitable for a thread migration context. The scenario we are talking about is a wide-opened one, where different machines running JikesRVM mutually exchange their `MobileThreads` without sharing the main memory. We introduced, therefore, a portable version of the scope descriptor, called `MobileFrame`, whose structure is reported in Figure 6. While the OSR implementation in JikesRVM uses an internal object of class `VM_NormalMethod` to identify the method of the frame, we cannot make such an assumption; the only way to identify that method is through the triplet

<*method name, method descriptor, full class name*>

that is universally valid. This triplet (represented by the three fields `methodName`, `methodDescriptor` and `methodClass` in Figure 6) is used to refer the method at the destination (e.g. its bytecode must be downloaded if not locally available yet), maybe after a local compilation. The bytecode index (i.e. the `bcIndex` field) is the most portable form to represent the return address of each method body and it is already provided in JikesRVM by default OSR. Finally, we have two arrays (i.e. the `locals` and `stack_operands` fields) that, respectively, contain the values of local variables (including parameters) and stack operands in that frame. These values are extracted from the physical frame at the specified bytecode index and converted into their corresponding Java types (`int`, `float`, `Object` references and so on). In addition, it must be pointed out that the `OSR_MobilityExtractor` class fixes up some problems that we run across during our implementation: here, we think it is worthwhile mentioning the problem of "uninitialized local variables". Default OSR extractor does not consider, in the JVM scope descriptor, those variables that are not active at the specified bytecode index. Nevertheless, these local variables have their space allocated in the stack and this fact should be taken into account when that frame is re-established at the destination.

```
class MobileFrame {
    /** Name of the method which adds this frame*/
    public String methodName;

    /** Method descriptor
     e.g. "(I)V" for a method
     getting an integer and returning void */
    public String methodDescriptor;

     /** Fully qualified method class
     (e.g."mypackage.myClass")*/
    public String methodClass;

     /** The bytecode index (i.e. return address)
     within this method*/
    public int bcIndex;

     /** The local bytecode-level local variable
     including parameters */
    public MobileFrameElement[] locals;

     /** The value of the stack operands at the
     specified bytecode index */
    public MobileFrameElement[] stack_operands;

    // methods and static fields omitted…
}
```

Figure 6. The main fields of the `MobileFrame` class

To summarize, in our mobility framework threads are serialized in a strong fashion: the `MobileThread` object is serialized as a regular object, while the execution state is transferred as a chain of fully serializable `MobileFrame` objects (produced by multiple invocations of the `extractSingleFrame()` method of Figure 5).

3.4 Resuming a migrated thread

The symmetrical part of the migration process is the creation, at the destination host, of a local instance of the migrated thread. This task is appointed to the service input-channel threads that listen for migratory threads coming from the network. The entire process has been summarized in Subsection 3.1, but in this one we are going to see how the thread is rebuilt in JikesRVM.

The first operation is creating a thread whose only task is to start execution and auto-suspend. This allows the infrastructure to safely reshape the current stack object of this thread, injecting one by one all the frames, belonging to the arrived thread. In more details, a new stack is allocated and it is filled with the thread-switch internal frames, taken from the auto-suspended thread. Then, every `MobileFrame` object is installed, in the same order as they were read from the socket stream (i.e. from the `Methodn()` to `run()`, looking at Figure 4). The brand-new stack is closed with the remaining preamble frames, again borrowed from the auto-suspended thread. The code in Figure 7 shows the above phases. Now, the new stack has been prepared and the context registers are properly adjusted (pointers are updated to refer to the new stack memory). This stack takes the place of the old stack belonging to the auto-suspended thread (the old one is discarded and becomes “garbage”). The new `MobileThread` object, with its execution state completely re-established, can be transparently resumed and continues from the next instruction.

```
void installStack() {
   // omitted auxiliary local variables
   /*1. First of all we have to compute the
   required space for the new stack to allocate*/
   for(int i=0;i<frameSet.size();i++){
       frame = (MobileFrame) frameSet.get(i);
       userFrameSpace+=frame.computeRealSize();
   }

   stackSpace=fixedFramesSizes[0] /*preamble*/ +
   userFrameSpace + fixedFramesSizes[1] /*thread-
   switch part*/;

   /*2. With the computed space, allocate a new
   stack*/
   byte[] newStack = MM_Interface.newStack(
                             stackSpace,false);

   /*3. Attach the top frames for suspension*/
   attachTopFrames(newStack);

   /*4. Install every MobileFrame read from the
   socket stream*/
   for(int i=0;i<frameSet.size();i++) {
       frame = (MobileFrame) frameSet.get(i);
       frame.installFrame(/*omitted parameters*/);
   }
   /*5. Copy the bottom frames to complete stack*/
   copyBottomFrames(newStack);
}
```

Figure 7. The stack installation phases in a code excerpt

3.5 Additional features

In this section, some additional features implemented in our JikesRVM-based framework are outlined. These features, even though not strictly essential for the purpose of pure thread migration implementation, are likewise important, because they characterize the programming paradigm proposed to the programmer.

3.5.1 Dealing with object references

The set of all the referenced objects of a thread has been prior defined as its *data space* [11] and, at any point during the execution, is composed of all the objects that can be reached by the thread through the call stack or through references to other objects. As concerns the stack, the space that the thread is supposed to carry with itself comprises all the objects pointed by the parameters and local variables of methods, together with those objects pushed on the operand stack of each frame in the stack.

Object references pose three kinds of problems, when designing a mobility framework: *(i) identifying object references* in the call stack of the thread; (ii) *collecting* some of these objects; *(iii) properly relocating the collected objects* according to some relocation strategy. As concerns the *identification* of object references in the call stack, it must be pointed out how we tackled this problem with a zero-overhead approach, thanks to the *type accurate garbage collectors* provided by JikesRVM. Traditional JVMs do not provide any information at runtime that can help inferring the types of data pushed on the thread’s call stack (i.e. local variables, stack operands and parameters). The only place where these types are known is the bytecode of the methods that pushed the data on the stack. Many systems [22, 21] take the easier way of modifying the Java compiler to store type information "somewhere", when a value is pushed on the stack by a bytecode instruction (e.g. maintaining a separate type stack for each thread). This approach requires significant modifications to the JVM and it introduces an important overhead on the bytecode execution, since a parallel type stack has to be managed. Another possible approach [5] delays such “type inference” until migration time.

JikesRVM uses *type-accurate* garbage collectors that build the so-called *reference maps* automatically at compile-time, unlike *conservative collectors*, which attempt somehow to infer whether a stack word is a reference or not [27]. These reference maps are periodical snapshots of the situation of references in each method frame, taken only at the above mentioned *yieldpoints*. Therefore, since our `MobileThreads` are allowed to migrate only in their migration points (which are also yieldpoints, as explained in Subsection 3.2), object references in the stack are safely identified and collected by the framework. These reference maps, which are used to speed up JikesRVM *type-accurate* garbage collectors, rid us of the need for more potentially intrusive “type inference” mechanisms required by many existing strongly mobile systems.

After their identification, *object references must be collected* for serialization. According to the above definition of data space, we thought natural to scatter fragments of the data space directly among the portable frames mentioned in the previous Subsection. Local and stack operands are transferred in two arrays of `MobileFrameElement` objects (refer to Figure 6). The conceptual structure of such elements and their relationship with a `MobileFrame` is graphically schematized in Figure 8. The `ref` field of the `MobileFrameElement` contains the reference to the object, to be serialized together with its owner `MobileFrame` object (non-reference values are stored in another specific field).

As for the *relocation of such objects*, it has been argued [11] that the conventional Java serialization is often not well suited to the requirements of mobile thread programmers. Some objects, such as I/O devices, cannot be blindly serialized; rather they have to be adjusted (e.g., being substituted by their local copies on the destination JVM). Furthermore, other objects require a “dedicated management” when they travel to another JVM heap. Consider, for instance, a `File` object, which is serializable but, when serialized, the binding to the file system object gets lost. If the programmer wants that a `File` object is available even from remote hosts over the network, she must be able to handle the migration/serialization, thus for example she can substitute the file “pointer” to a local file with a network file (using for instance NFS). Such issues can be dealt with exploiting, for instance, the Java serialization flexibility: using the `Externalizable` interface the programmer can customize the serialization of her objects [24]. It is also frequent to have objects shared by multiple threads in the system: in such cases, currently the migration mechanism simply serializes the object toword destination and does not affect other threads referencing the same object (i.e. the object stays alive in the source heap). We guess some new programming language constructs would be desirable, in order to place constraints on specific objects, preventing inconsistency situations after thread migration (e.g. a sort of “object pinning”).

Though crucial for the successful adoption of the mobile code paradigm (they are present in weakly mobile systems as well), such issues are not of concern for this technical paper.

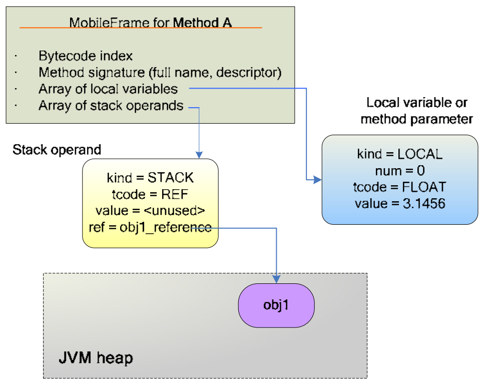


Figure 8. A schematic representation of a `MobileFrame`

3.5.2 Multi-threaded migration

Creating multiple, concurrent, related threads with mutual interactions becomes, in many distributed computing scenarios, a real must: in these cases, the concept of execution unit [11] can be generalized to comprise multi-threaded applications. The trickiest implications of this extended concept come from the scattered nature of the state to be captured and migrated. Thus, we have enhanced the presented mobility infrastructure to support thread groups serialization too.

Starting from the per-thread serialization protocol, we examined what serializing a group implies and experimented a simple recursive protocol for this. Thread groups (i.e. `java.lang.ThreadGroup`) are useful for two reasons: they allow the programmer to manipulate many threads by calling a single method, and they provide the basis that Java security mechanism uses to interact with threads [16]. Groups are structured hierarchically as a tree with a "main" root and unlimited levels of subgroups. Thus, chosen a group as the *serialization root*, a `migrate()` method recursively serializes its sub-tree of groups and every thread within each group. The sub-tree is moved from the source JVM group hierarchy and attached to the destination JVM hierarchy. Thus, if a mobile multithreaded application has to be migrated on multiple machines (e.g. a GUI Java application where we can have an event thread, timers and callback threads), the programmer has just to follow the standard concept of Java thread groups, creating siblings threads belonging all to the same `MobileThreadGroup`.

Group migration is carried out as a sort of *synchronized migration*, with a *synchronization barrier* established when a migration is requested on a certain group. All threads belonging to that group or to children subgroups synchronize themselves with the barrier (proactively or reactively, as seen in Subsection 3.2) and, when all threads have reached the synchronization point, the migration begins. Moving from the root to the leaves, each group object is serialized, together with its owned threads; then, recursively all its direct subgroups are serialized in their turn and thus migrated. In order to carefully manage this distributed state, the relationships between threads and groups must be accounted for. For this purpose, we have chosen a symbolic naming for threads and groups in the serialized sub-tree, using unique numeric identifiers (id). Therefore, each reference within the sub-tree is temporary substituted by the internal id of the referenced element (thread or group). When deserialization is performed, ids are re-converted to references with the help of a specific hash-map. Threads belonging to a `MobileThreadGroup` can only migrate as a group and never separately and references to such threads in other external threads are set to null, because they are no longer locally available. As for shared objects among migrating and non migrating threads, it must be underlined that, using Java serialization for their migration, they are actually "cloned" remotely. As already mentioned in Subsection 3.5.1, we deem all other possible object migration strategies bound to the specific user application requirements.

4. PERFORMANCE AND EVALUATION

At the current stage of our project (whose code is available at http://www.agentgroup.unimore.it), the thread serialization mechanism, discussed so far, has been successfully tested, focusing mainly on the times needed for state capturing and restoring. We made, therefore, some performance tests to discover possible bottlenecks and evaluate the cost of each migration phase. We wrote a benchmark based on the Fibonacci recursive algorithm, just to see the variation of migration times with respect to an increasing number of frames into the stack. The times measured are expressed in seconds and are average values computed across multiple runs, on a Pentium IV 3.4 GHz, 1GB RAM,JikesRVM release 2.4.1. Some sample times (taken with a number of 5, 15 and 25 frames) are listed in Table 1 and Table 2 and they demonstrate very graceful time degradation. The times have been conceptually divided into two phases, where Table 1 refers to the thread serialization phase, while Table 2 refers to the corresponding de-serialization phase at the arrival host (the transfer time through the network has not been considered significant here and so it is not reported).

Considering how these times are partitioned among the different phases of the thread serialization, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state, while the frame extraction mechanism (i.e. the core of our entire facility) has very short times instead.

	5 frames	**15 frames**	**25 frames**
OSR Frame capturing	1.78E-5	1.89E-5	1.96E-5
State building	3.44E-5	3.75E-5	3.43E-5
Pure serialization	2.49E-3	7.32E-3	1.50E-2
Overall times	2.54E-3	7.38E-3	1.51E-2

Table 1. Evaluated times for thread serialization (sec.)

The same bottleneck due the Java serialization may be observed in the de-serialization of the thread. In the latter case, however, we have an additional foreseeable time in the stack installation phase, since the system has often to create a new thread and compile the methods for the injected frames. These performance bottlenecks can be further minimized, perhaps using externalization to speed up the serialization of the thread state [24].

	5 frames	15 frames	25 frames
Pure de-serialization	4.46E-3	5.33E-3	7.06E-3
State rebuilding	5.45E-4	5.27E-4	5.06E-4
Stack installation	1.53E-3	1.60E-3	1.71E-3
Overall times	6.54E-3	7.46E-3	9.28E-3

Table 2. Evaluated times for thread rebuilding (sec.)

In Subsection 3.1, a JIT compiler extension was described that allowed us to perform reactive migration, in addition to the simpler proactive case. We said that, in that cases, migration points are installed instead of traditional yieldpoints in some chosen method bodies. This has two negligible costs:

1. on *thread execution time*, since migration points are taken only if a thread switch is requested (they are in fact a subset of yieldpoints). However, such an approach does not suffer from the slowdown of many application-level approaches reported in Subsection 2.3.2: many of those systems inject checkpoint instructions into the bytecode, to trace the execution state and let the thread migrate only in predefined points. We do not, instead, insert any additional checkpoint in the code, but simply extend the functionality of normal pre-existent JikesRVM yieldpoints.
2. on *method JIT compilation time*; this is due to an additional test to be performed on the candidate method to JIT compilation. Referring again to Subsection 3.1, the extended `genThreadSwitchTest()` method performs three simple necessary tests, to determine if the method needs the insertion of our migration points or the old JikesRVM yieldpoints.

No other overhead are imposed on JikesRVM normal performances. In addition, group migration (beside the time needed for each thread to reach the synchronization barrier) has essentially identical measured times, since it uses the same single-thread migration mechanism.

5. FUTURE WORK

Additional features can be, of course, implemented to extend our thread mobility framework in the future. First of all, the optimized compiler is not fully supported yet. OSR can extract the JVM scope descriptor even from optimized method frames, but this requires some cooperation from the optimizing compiler to generate mapping information needed to correctly interpret the structure of the optimized frame: in fact, while baseline frames have a fully predictable layout, the same is not true for optimized ones where local variables can be allocated into machine registers, pieces of code can be suppressed or inlined, and so forth. For these reasons, the optimizing compiler choices some points in the code where OSR can occur and, just for these points, maintains all the necessary mapping information. Such points, called *OSR Points*, do not include "method call sites" and for that reason our serialization system cannot capture optimized frames yet. Nevertheless, we are aware of a project by Krintz et al.[20] trying to present a more general-purpose version of OSR that is more amenable to optimizations than the current one. The improvement descending from this work will be exploited to perform a more complete thread state capturing, even in presence of code optimizations.

Future work includes also a comparison with other proposed thread migration systems, to improve our performance evaluation understanding and identify possible undetected bottlenecks. Finally we are currently working to port the implemented code also on PPC architectures (JikesRVM is available also for this processor), allowing the migration of a thread among heterogeneous platform as well.

6. CONCLUSIONS

This paper has presented our framework that extends the facilities provided by the IBM JikesRVM in order to support Java thread strong migration. Thanks to its modular design and its minimally intrusive nature, the developed framework can be easily adopted in distributed application developments, provided that a recent OSR-enabled version of JikesRVM is installed in the system (it can be classified as a midway approach between the described JVM-level and the application-level approaches). Users do not have to download a modified, untrustworthy, version of JikesRVM, but can import the implemented mobility package into their code and execute it on their own copy of JikesRVM. Moreover, thanks to the support for thread-group migration, even complex multi-threaded applications can become fully mobile.

Acknowledgements. Work supported by the European Community within the EU FET project "CASCADAS".

REFERENCES

[1] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, S. Smith, *Implementing Jalapeño in Java*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), Denver, Colorado, November 1, 1999

[2] B.Alpern, C.R. Attanasio, D. Grove and others, *The Jalapeno virtual machine*, IBM System Journal, Vol. 39, N°1, 2000

[3] M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney, *Adaptive Optimization in the Jalapeño JVM*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 15-19, 2000

[4] A. Balachandran, G. M. Voelker, and P. Bahl, *Wireless Hotspots: Current Challenges and Future Directions*, in Mobile Networks and Applications Journal, Springer Science Publishers, pp. 265-274, 2005

[5] S. Bouchenak, D. Hagimot, *Pickling Threads State in the Java System*, Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000) Mont-Saint-Michel/Saint-Malo, France, June 2000

[6] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma and F. Boyer, *Experiences Implementing Efficient Java Thread Serialization*, Mobility and Persistence", I.N.R.I.A., Research report n°4662, December 2002

[7] G. Burke, J.Choi, S. Fink, D.Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, *The Jalapeno Dynamic Optimizing Compiler for Java*, ACM Java Grande Conference, June 1999

[8] C. Chambers, *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, PhD thesis, Stanford University, Mar. 1992. Published as tech. report STAN-CS-92-1420

[9] G. Cabri, L. Leonardi, F. Zambonelli, *Weak and Strong Mobility in Mobile Agent Applications*, Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK), April 2000

[10] S. Fink, F. Qian, *Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement*, International Symposium on Code Generation and Optimization San Francisco, California, March 2003

[11] A. Fuggetta, G. P. Picco, G. Vigna, *Understanding Code Mobility*, IEEE Transactions on Software Engineering, Vol 24, 1998

[12] S. Funfrocken, *Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the state of Java Programs)*, 2nd International Workshop on Mobile Agents 98 (MA'98), Stuttgart, Germany, Sep. 1998

[13] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification, second edition*, SUN Microsystem

[14] T. Illmann, T. Krueger, F. Kargl, M. Weber, *Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture*, Proceedings of the 5th International Conference on Mobile Agents, Atlanta, Georgia, USA, December 2001

[15] M. Kim, D. Kotz and S. Kim. *Extracting a mobility model from real user traces*, In Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Barcelona, Spain, April, 2006

[16] S. Oaks, H. Wong, *Java Threads, 2nd edition*, Oreilly, 1999

[17] *The OpenMosix Project*, http://openmosix.sourceforge.net/

[18] T. Sakamoto, T. Sekiguchi, A. Yonezawa, *A Bytecode Transformation for Portable Thread Migration in Java*, 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Sep. 2000

[19] T. Sekiguchi, A. Yonezawa, H. Masuhara, *A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation*, 3rd International Conference on Coordination Models and Languages, Amsterdam, The Netherlands, Apr. 1999

[20] S. Soman, C. Krintz, *Efficient, General-Purpose, On-Stack Replacement for Aggressive Program Specialization*, University of California, Santa Barbara Technical Report #2004-24

[21] T. Suezawa, *Persistent Execution State of a Java Virtual Machine*, ACM Java Grande 2000 Conference, San Francisco, CA, USA, June 200

[22] A. Acharya, M. Ranganathan, J. Saltz, *Sumatra: A Language for Resource-aware Mobile Programs*. 2nd International Workshop on Mobile Object Systems (MOS'96), Linz, Austria, 1996

[23] N. Suri et al., *An Overview of the NOMADS Mobile Agent System*, Workshop On Mobile Object Systems in association with the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Cannes, France, 2000

[24] Sun Microsystems. *Improving Serialization Performance with Externalizable*, http://java.sun.com/developer/TechTips/txtarchive/2000/Apr00_StuH.txt

[25] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, P. Verbaeten, *Portable support for Transparent Thread Migration in Java*, 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Switzerland, Sep. 2000

[26] X. Wang, *Translation from Strong Mobility to Weak Mobility for Java*, Master's thesis, The Ohio State University, 2001

[27] P.R.Wilson, *Uniprocessor Garbage Collector Techniques*, in the Proceedings of the International Workshop on Memory Management (IWMM92), St. Malo, France, September 1992

[28] W. Zhu, C. Wang, F. C. M. Lau, *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*. IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September

Juxta-Cat: A JXTA-based platform for distributed computing[*]

Joan Esteve Riasol
Computing Laboratory
Informatics Faculty of Barcelona
Campus Nord, C5
C/Jordi Girona 1-3
08034 Barcelona, Spain
jesteve@fib.upc.edu

Fatos Xhafa
Dept. of Languages and Informatics Systems
Polytechnic University of Catalonia
Campus Nord, Ed. Omega
C/Jordi Girona 1-3
08034 Barcelona, Spain
fatos@lsi.upc.edu

ABSTRACT

In this paper we present a JXTA-based platform, called Juxta-CAT, which is an effort to use the JXTA architecture to build a job execution-sharing distributed environment. The Juxta-CAT platform has been deployed in a large-scale, distributed and heterogeneous P2P network, based on open JXTA protocols. The goal of our platform is to create a shared Grid where client peers can submit their tasks in the form of java programs stored on signed jar files and are remotely solved on the nodes of the platform. Thus, the main goal of our platform is giving direct access to resources and sharing of the computing resources of nodes, in contrast to other well-known P2P systems that only share hard disk contents.

The architecture of our platform is made up of two types of peers: common client peers and broker peers. The former can create and submit their requests using a GUI-based application while the later are the administrators of the Grid, which are in charge of efficiently assigning client requests to the Grid nodes and notify the results to the owner's requests. To assure an efficient use of resources, brokers use an allocation algorithm, which can be viewed as a price-based economic model, to determine the best candidate node to process each new received petition. The implementation and design of peers, groups, job and presence discovery, pipe-based messaging, etc. are developed using the latest updated JXTA libraries (currently release 2.3.7) and JDK 1.5 version.

Several types of applications arising from different fields such as scientific calculations, simulations, data mining, etc. can be solved in Juxta-CAT. To create a suitable demo scenario and test the proposed platform, we have joined and used the PlanetLab platform (http://www.planet-lab.org/). Juxta-CAT Project and its official web site have been hosted in Java.NET community at https://juxtacat.dev.java.net.

[*]Research partially supported by ASCE Project TIN2005-09198-C02-02 and Project FP6-2004-IST-FETPI (AEOLUS).

1. INTRODUCTION AND MOTIVATION

The rapid development of Internet and other new technologies has yielded to new paradigms of distributed computing, among others the *Computational Grids* and *P2P systems*. Computational Grids were introduced by Foster and other researchers in late '90 [7, 8, 6]. The main focus of this new distributed paradigm was that of providing new computational frameworks by which geographically distributed resources are logically unified as a computational unit. As pointed out by Foster et al. Computational Grid is *"... a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically depending on their availability, capability, performance, cost, and users' QoS requirements."* Grid computing motivated the development of large scale applications that benefit from the large computing capacity offered by the Grid. Thus, several projects such as NetSolve [4], MetaNeos Project (Metacomputing environments for optimization), and applications for Stochastic Programming [9], Optimization Problems [15], to name a few, used grid computing.

On the other hand, P2P systems [13, 11, 2] appeared as the new paradigm after client-server and web-based computing. P2P systems are quite known due to popular P2P systems such as Napster, Gnutella, FreeNet and others. One of the main characteristics of such systems is file sharing among peers. Due to this, and in order to stress the difference between computational grids and P2P systems, Foster et al. remarked that *"... the sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource brokerage strategies emerging in industry, science, and engineering"*

It should be noted however that since the appearance of first grid systems and P2P systems, the grid computing and P2P computing has evolved so that both paradigms have many characteristics in common, among which we distinguish the *distributed computing.* It is precisely this characteristic that we explore in this work, that is, the use of resource sharing for problem solving in dynamic environments. Grid systems has shown to be very useful for real world applications, indeed, several types of grids has appeared such as *Compute Grids, Data Grids, Science Grids, Access Grids, Bio Grids, Sensor Grids, Cluster Grids, Campus Grids, Tera Grids, and Commodity Grids*, among others. The success of computational grids could be explained by the development of several middleware, such as Globus, MPI-Grid2 and Condor-G that facilitate the development of grid-based applications. On the P2P side, the improvement of P2P protocols is enabling the development of P2P applications others than the well-known file-sharing applications. However, there is still few work to bring P2P system to real word applications, mainly due to the lack of robust P2P platforms that would allow the deployment of large P2P systems. This is precisely the motivation of this work. We propose a P2P platform, called Juxta-CAT, developed using JXTA architecture.

The goal of the Juxta-CAT platform is to create a shared Grid where client peers can submit their tasks in the form of java programs stored on signed jar files and are remotely solved on the nodes of the platform. Thus the Juxta-CAT is a real peer-to-peer environment that represents a workspace structured as a grid, where peer clients can share their resources of CPU, memory and hard disk for solving their problems.

The architecture of our platform is made up of two types of peers: common *client peers* and *broker peers*. The former can create and send their requests using a GUI-based application while the later are the administrators of the Grid, which are in charge of efficiently assigning client requests to the Grid nodes and notify the results to the owner's requests. One key issue in such systems is the flexible and efficient use of resources [5, 3, 14, 12] in order to benefit from the computing capacity as efficiently and economically possible. To this end, the brokers in our platform use an allocation algorithm, which can be viewed as a simple price-based economic model, to determine the best candidate peers to process each new received petition. The implementation and design of peers, groups, job and presence discovery, pipe-based messaging, etc. are developed using the latest updated JXTA libraries (currently release 2.3.7) and JDK 1.5 version. Our proposal extends and improves several existing features of the JXTA, especially those related to the management of presence mechanism and the pipe service system.

Several types of applications arising from different fields such as scientific calculations, simulations, data mining, etc. can be solved using Juxta-CAT. To create a suitable demo scenario and test the proposed platform, we have joined and used the PlanetLab platform (http://www.planet-lab.org/). In this stage of the work we have considered some simple applications (e.g. computing Π number with high precision), namely applications that are composed of many independent tasks that can be simultaneously submitted and solved in the grid. Using this type of applications we were able to test the robustness of the Juxta-CAT and measure its performance as regards the speed-up of computation. Our experimental results show the feasibility of the Juxta-CAT for developing large-scale applications as we have deployed the grid applications in a real P2P network based on PlanetLab nodes.

Juxta-CAT Project and its official web site have been hosted in Java.NET at https://juxtacat.dev.java.net and has also been published in the JXTA Developer Spotlight of March 2006 at http://www.jxta.org.

The paper is organized as follows. We give in Sect. 2 some preliminaries on JXTA protocols that we have used in the development of Juxta-CAT. The architecture of the Juxta-CAT platform is given in Sect. 3. In Sect. 4 we give some of the implementations issues, the improvements on JXTA protocols as well as the allocation algorithm used by brokers. The evaluation of the Juxta-CAT platform and some computational results are given in Sect. 5. We conclude in Sect. 6 with some remarks and point out further work.

2. OVERVIEW ON BASIC JXTA PROTOCOLS

JXTA technology is a set of open protocols proposed by Sun Microsystems that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner (see e.g. [2, 10]). JXTA peers create a virtual network in which any peer can interact with other peers and resources directly even when some of the peers and resources are behind fire-walls and NATs or when they are on different network transports.

The current specification of this architecture determines the management of the clients (peers) of the developed networks.

- Discover peers and resources on the network even across fire-walls
- Share files with peers across the network
- Create your own group of peers of devices across different networks
- Communicate securely with peers across public networks

One important advantage of JXTA is its platform independence as regards:

- *Programming language*: Implementations of JXTA protocols are found in Java, C, C++ and other languages.
- *Transport protocol*: It is not required to use an specific net structure. P2P applications can be developed over TCP/IP, HTTP, Bluetooth and other transports.
- *Security*: Developers are free to manage the security issues of the developed platform.

The updated core libraries for each implementation are found at the official JXTA page at http://www.jxta.org. For the purposes of the development of Juxta-CAT platform, we needed to only work with the J2SE version of JXTA.

The JXTA protocols have been structured as a *working stack*, as can be seen in Fig. 1. It should be noticed that JXTA protocols are independent among them. A definition of a JXTA peer does not require implementing the set of all protocols in order to participate in the network. For instance, a node may need not statistical information about the other members of the peergroup, therefore it is not necessary for that node to implement the Protocol of Information or a node could have previously stored in its memory the set of peers or services to work with, and hence it does not need the additional protocol of resource discovery.

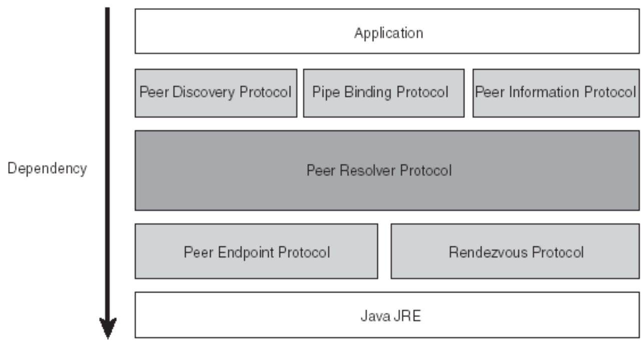


Figure 1: JXTA Architecture.

3. JUXTA-CAT ARCHITECTURE

In this section we describe the basic aspects of the Juxta-CAT platform. Before starting the design of the platform we carefully identified the possible roles that each peer participant was going to play in the Juxta-Cat platform. The importance of a clear specification of the peer roles was crucial to the architecture of the platform. Indeed, each role represents a "subsystem" of the platform requiring a different treatment during design and implementation stages. From now on, we use the term *peer* to denote the machine or node on the Juxta-Cat platform. As we show next, peers can execute either a client or a broker and hence we speak of *client peers* and *broker peers*.

We will also speak of request, petition, job or task indistinctly to denote a task submitted to the grid. To be precise, the request will contain all the information that a grid node, which receives the request, needs for solving a task.

3.1 Client peers

Client peers are the end users of the Juxta-CAT. Client peers are obtained by downloading and installing the application from the official page of Juxta-CAT. Once the machine is "converted" into a client peer, on the one hand, the user will connect to the peer-to-peer network and submit execution requests to their peergroup partners. One the other hand, client peers will also be able to process received requests sent to it by other nodes through the brokering and notify them the result of the request, once it is completed (see Fig. 2 for an UML diagram representation.)

Therefore, a client peer plays two possible roles as regards the resolution requests:

(a) submitting job requests to the grid and receiving the results.

(b) receiving job requests submitted to the grid by other client peers and notifying the result to the corresponding owners of the requests.

Note that, while submitting job requests to the grid, a client peer can also decide whether it includes itself as a possible candidate peer to receive the proper submitted requests or not.

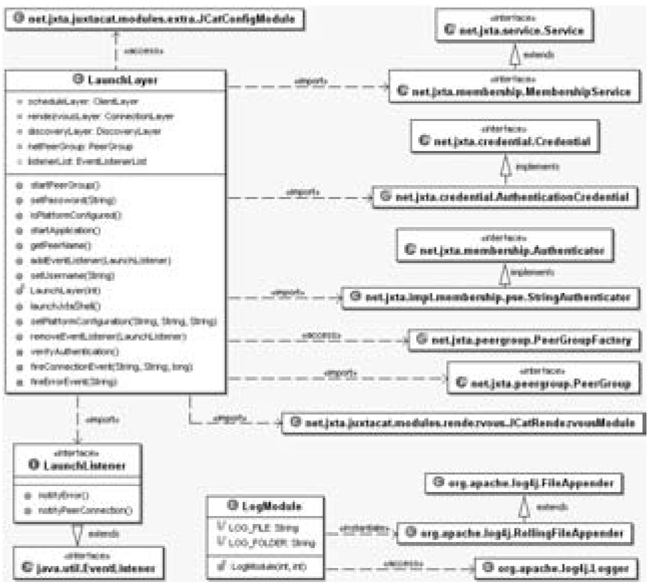

net.jxta.juxtacat.modules.extra.JCatConfigModule
«access»
LaunchLayer
scheduleLayer: ClientLayer
rendezvousLayer: ConnectionLayer
discoveryLayer: DiscoveryLayer
netPeerGroup: PeerGroup
listenerList: EventListenerList
startPeerGroup()
setPassword(String)
isPlatformConfigured()
startApplication()
getPeerName()
addEventListener(LaunchListener)
setUsername(String)
LaunchLayer(int)
launchJxtaShell()
setPlatformConfiguration(String, String, String)
removeEventListener(LaunchListener)
verifyAuthentication()
fireConnectionEvent(String, String, long)
fireErrorEvent(String)
«interface»
net.jxta.service.Service
extends
«import»
«interface»
net.jxta.membership.MembershipService
«interface»
net.jxta.credential.Credential
implements
«import»
net.jxta.credential.AuthenticationCredential
«interface»
net.jxta.membership.Authenticator
implements
«import»
net.jxta.impl.membership.pse.StringAuthenticator
«access»
net.jxta.peergroup.PeerGroupFactory
«import»
«interface»
net.jxta.peergroup.PeerGroup
«import»
«import»
net.jxta.juxtacat.modules.rendezvous.JCatRendezvousModule
«interface»
LaunchListener
notifyError()
notifyPeerConnection()
extends
«interface»
java.util.EventListener
LogModule
LOG_FILE: String
LOG_FOLDER: String
LogModule(int, int)
org.apache.log4j.FileAppender
extends
«instantiate»
org.apache.log4j.RollingFileAppender
«access»
org.apache.log4j.Logger

Figure 2: UML diagram of the Juxta-CAT Client.

3.2 Broker peers

Broker peers are the *governors* of the request allocation in Juxta-Cat (see Fig. 3 for an UML diagram representation.)

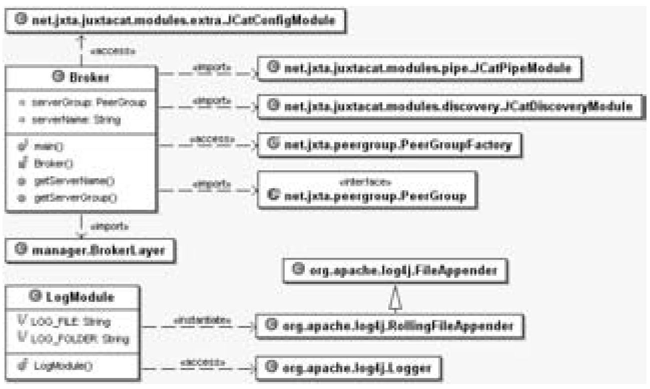

net.jxta.juxtacat.modules.extra.JCatConfigModule
«access»
Broker
serverGroup: PeerGroup
serverName: String
main()
Broker()
getServerName()
getServerGroup()
«import»
net.jxta.juxtacat.modules.pipe.JCatPipeModule
«import»
net.jxta.juxtacat.modules.discovery.JCatDiscoveryModule
«access»
net.jxta.peergroup.PeerGroupFactory
«import»
«interface»
net.jxta.peergroup.PeerGroup
«import»
manager.BrokerLayer
org.apache.log4j.FileAppender
LogModule
LOG_FILE: String
LOG_FOLDER: String
LogModule()
«instantiate»
org.apache.log4j.RollingFileAppender
«access»
org.apache.log4j.Logger

Figure 3: UML diagram of the Juxta-CAT Broker.

Brokers act like *bots* of the network: they are connected to the to P2P platform and are in charge of receiving and allocating the requests sent by clients of the peergroup. Whenever a broker receives a request, it explores the state of the rest of nodes currently connected to the network, examining their working and connection statistics. Then, it uses this historical/statistical data to select, according to an simple price-based economic model, the best candidate peer for processing that request.

Once the best node, for a given request, is selected, the broker sends to that node a message with the information needed for the processing the request. If there were some error in sending the message, or if there were no nodes available to receive the requests (a request not necessarily is processable by all nodes) then the broker adds the request to the queue of pending requests. Thus, each broker maintains and manages its own pending request queue, which is updated according to the state of the network and the arrival on new requests (see Fig. 4).

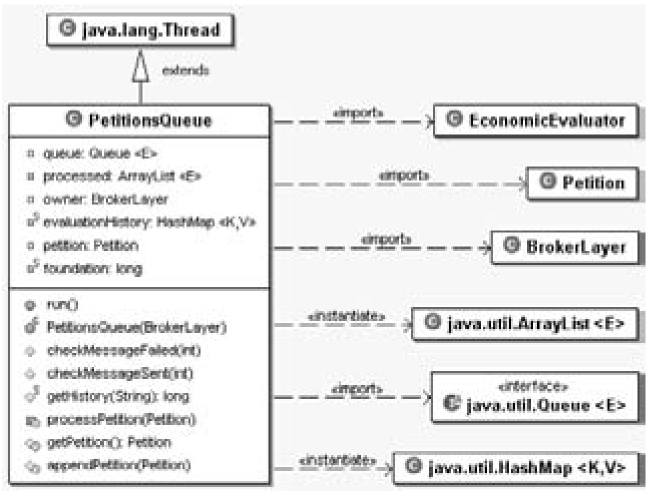


Figure 4: UML diagram of the requests queue maintained by the Broker.

4. IMPLEMENTATION ISSUES

The design pattern used for Juxta-CAT is an integration of the Model-View-Controller and Observer patterns. We have thus three independent modules or layers: *View*, *Model* and *Control* incorporating the *observer* pattern to communicate the results between different layers (from Control to Model and from Model to View). We give this organization of Juxta-CAT in layers in Fig. 5 (see also Fig. 6).

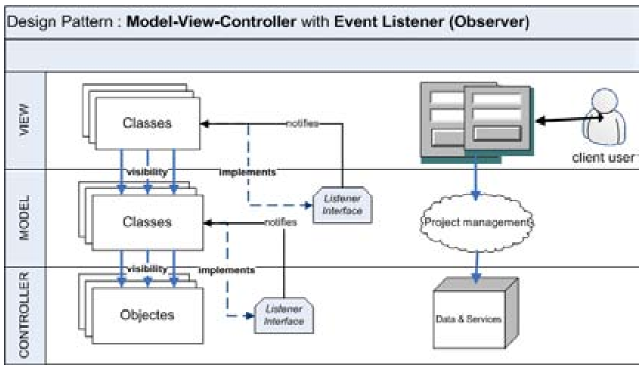
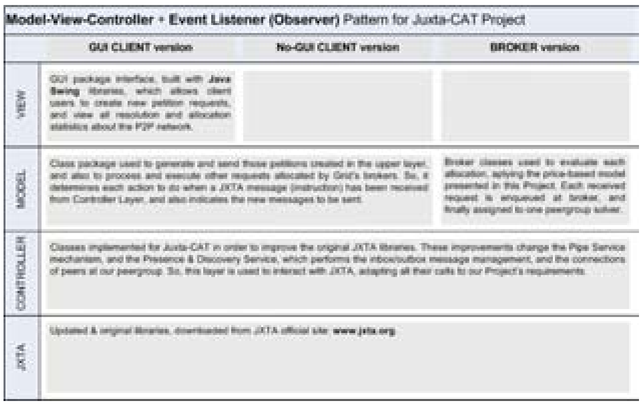


Figure 5: JXTA architecture (MVC+Observer).

Further, as can be seen from Fig. 6, we decided to implement two types of client peers: GUI client peers and CMD client peers. The former (implemented using Java Swing)



Model-View-Controller + Event Listener (Observer) Pattern for Juxta-CAT Project

	GUI CLIENT version	No-GUI CLIENT version	BROKER version
VIEW	GUI package interface, built with **Java Swing** libraries, which allows client users to create new petition requests, and view all resolution and allocation statistics about the P2P network.		
MODEL	Class package used to generate and send those petitions created in the upper layer, and also to process and execute other requests allocated by Grid's brokers. So, it determines each action to do when a JXTA message (instruction) has been received from Controller Layer, and also indicates the new messages to be sent.		Broker classes used to evaluate each allocation, aplying the price-based model presented in this Project. Each received request is enqueued at broker, and finally assigned to one peergroup solver.
CONTROLLER	Classes implemented for Juxta-CAT in order to improve the original JXTA libraries. These improvements change the Pipe Service mechanism, and the Presence & Discovery Service, which performs the inbox/outbox message management, and the connections of peers at our peergroup. So, this layer is used to interact with JXTA, adapting all their calls to our Project's requirements.		
JXTA	Updated & original libraries, downloaded from JXTA official site: **www.jxta.org**		

Figure 6: Juxta-CAT architecture (layer description).

allows the peer to have both functions (a) and (b) (see Subsect. 3.1), that is, submitting and receiving requests, while the later can only receive requests. In this way, any user can contribute his/her resources to the platform even when he/she is not interested in submitting requests to the grid.

During the development of Juxta-CAT, it has been necessary to improve the original JXTA libraries regarding the presence mechanism and pipe service system. JXTA is an open specification that only offers a standard mechanism, based on concrete but adaptable protocols, to build very basic peer-to-peer applications. In a certain sense, these libraries are just the starting point for the definition of a basic skeleton of the final application.

4.1 Improvements to JXTA protocols

In the case of Juxta-CAT platform, it is very important to maintain and efficiently manage the updated publication of the presence information and statistics as well as its publication to the nodes. The existing mechanisms offered by JXTA did not match our requirements. Therefore, it was necessary to change part of the implementation of the *Discovery Service* and *Pipe Service* of JXTA to adapt them to the requirements of Juxta-CAT. We deal with these changes in next paragraphs.

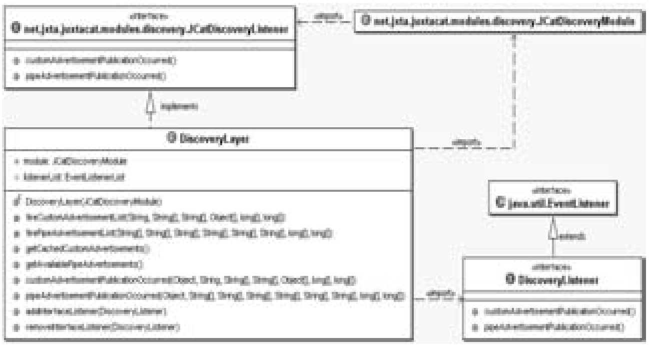
Improvement of the JXTA Discovery Service. Originally, JXTA maintains the peergroup information by the notification of presence that each node publishes in the cache of other nodes. But this procedure is not automatic, therefore, if we do not worry to refresh this information to the Discovery Service of Resources, we will never be able to guarantee that the collected data are updated.

Therefore, Juxta-CAT will have its own process, a Java Thread, that periodically updates the information contained in his local cache and trying to diminish the data traffic of the network. In this way, the discovery service in Juxta-CAT follows essentially these steps:

- Periodically publish the presence advertisement in its own local cache using the method `publish()` of the `DiscoveryService`.
- Publish remotely this advertisement in the cache of the rendezvous node of the peergroup (`remotePublish()` method).

- Send a discovery query to the peergroup asking for other presence advertisements.
- Copy responses to local cache and delete those that have expired.

(See also Fig. 7 for an UML diagram representation.)



net.jxta.juxtacat.modules.discovery.JCatDiscoveryListener
net.jxta.juxtacat.modules.discovery.JCatDiscoveryModule
implements
DiscoveryLayer
java.util.EventListener
extends
DiscoveryListener

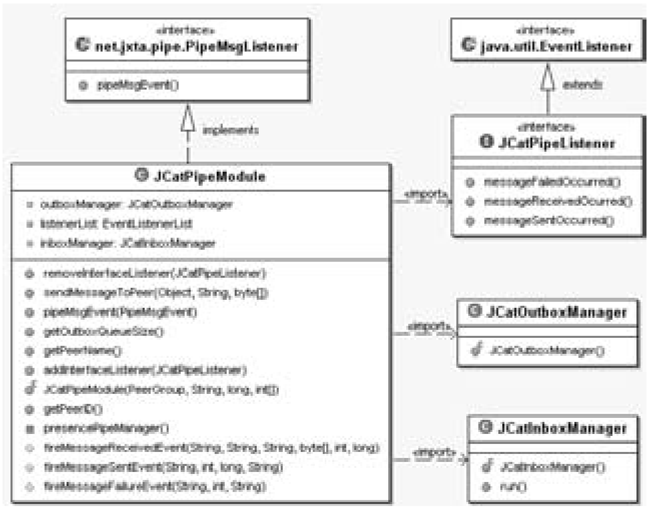
Figure 7: UML diagram of the Juxta-CAT Discovery Service manager.

Improvement of Pipe Service. We explain next how is improved the Pipe Service system that performs communication between nodes of a JXTA network. This is certainly one of the most important aspects of the Juxta-CAT. The main observation here is that JXTA Pipe Service doesn't check whether a message has been successfully delivered to its destination. In case of failure, JXTA doesn't attempt a new delivery of the message. Clearly, there is no timeout control as regards message delivery.

In order to send messages or data between peers, JXTA uses a pipe mechanism. A pipe is a virtual communication channel established between two processes. A computer connected to the network can open, at transport level, as many pipes as its operating system permits. These channels act as data links between the two communication points. Different programming languages such as C++ or Java have specific libraries with the necessary system calls for managing pipes with opening, reading, writing and closing operations of the connections.

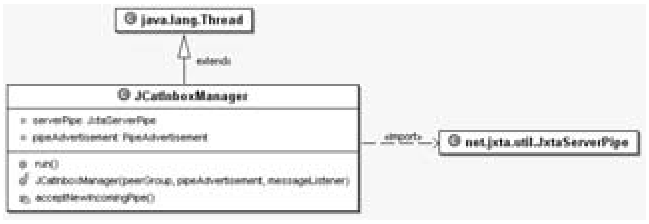
Next we briefly describe how we solved the above mentioned problems of the JXTA Pipe System resulting thus in an improved Pipe Service (see Fig. 8).

- *Presence through Pipe Advertisement*: a Juxta-CAT node declares, when its launched, an advertisement with the information relative to the user: role (client peer vs broker peer) within the grid, IP address, node name, and the unique JXTA identifier. This Advertisement is distributed by the network through Discovery Service, notifying to the rest of users of the grid the address of the pipe.
- *Inbox messages control* (see Fig. 9 for an UML diagram representation). Each node creates one pipe for receiving messages. This inbox pipe is managed by an independent thread that notifies to the superior layers of the system all the messages that are sent to the node.



«interface»
net.jxta.pipe.PipeMsgListener
pipeMsgEvent()
implements
«interface»
java.util.EventListener
extends
«interface»
JCatPipeListener
messageFailedOccurred()
messageReceivedOcurred()
messageSentOccurred()
JCatPipeModule
outboxManager: JCatOutboxManager
listenerList: EventListenerList
inboxManager: JCatInboxManager
removeInterfaceListener(JCatPipeListener)
sendMessageToPeer(Object, String, byte[])
pipeMsgEvent(PipeMsgEvent)
getOutboxQueueSize()
getPeerName()
addInterfaceListener(JCatPipeListener)
JCatPipeModule(PeerGroup, String, long, int[])
getPeerID()
presencePipeManager()
fireMessageReceivedEvent(String, String, String, byte[], int, long)
fireMessageSentEvent(String, int, long, String)
fireMessageFailureEvent(String, int, String)
«import»
JCatOutboxManager
JCatOutboxManager()
JCatInboxManager
JCatInboxManager()
run()

Figure 8: UML diagram of the package net.jxta.juxtacat.modules.pipe.*.



java.lang.Thread
extends
JCatInboxManager
serverPipe: JxtaServerPipe
pipeAdvertisement: PipeAdvertisement
run()
JCatInboxManager(peerGroup, pipeAdvertisement, messageListener)
acceptNewIncomingPipe()
«import»
net.jxta.util.JxtaServerPipe

Figure 9: UML diagram of the Juxta-CAT Inbox manager.

- *Outbox messages control* (see Fig. 10 for an UML diagram representation). In order to assure that the messages are successfully delivered to their addresses, we have developed a managing system based on queues. Each queue belongs to a possible destination node of our messages. When we want to send a message, we add it to its respective queue. Periodically, the manager sends simultaneously, according to the order of messages in the sequence, all the messages that have been kept in each queue. If a connection establishment fails, the message returns to the first position of the destination queue. A queue without pending messages to be sent remains blocked, and it does not consume resources nor memory of the Java Virtual machine.

- *Timeout Window.* A timeout window is used while trying to establish a connection. Each attempt to connect with a remote pipe through its advertisement has

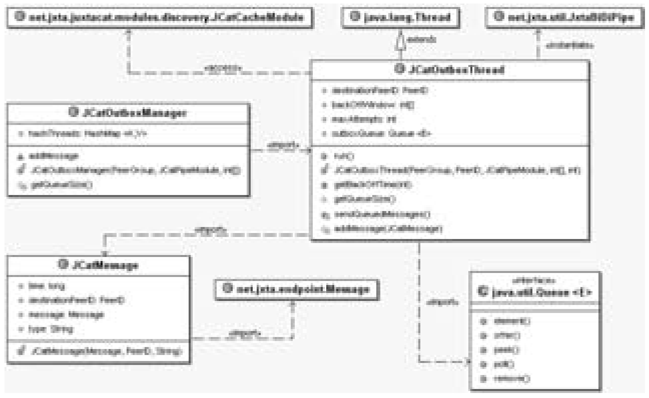

net.jxta.juxtacat.modules.discovery.JCatCacheModule
java.lang.Thread
net.jxta.util.JxtaBiDiPipe
JCatOutboxThread
JCatOutboxManager
JCatMessage
net.jxta.endpoint.Message
java.util.Queue <E>

Figure 10: UML diagram of the Juxta-CAT Outbox manager.

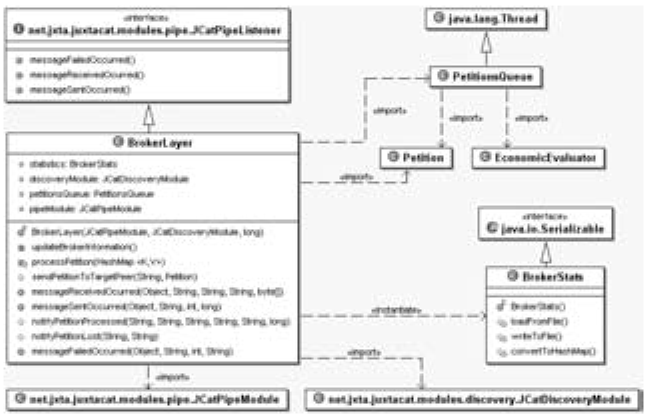

net.jxta.juxtacat.modules.pipe.JCatPipeListener
java.lang.Thread
PetitionsQueue
BrokerLayer
Petition
EconomicEvaluator
java.io.Serializable
BrokerStats
net.jxta.juxtacat.modules.pipe.JCatPipeModule
net.jxta.juxtacat.modules.discovery.JCatDiscoveryModule

Figure 11: UML diagram of the allocation model used by brokers of Juxta-CAT.

a timeout limit. We have defined a time interval, in a way that any failed connection retries the delivery with an increased timeout. We give in Table 1 the values for the timeout window used in Juxta-CAT.

Table 1: Timeout Window for sending and receiving.

Tries	1	2	3	4	5	> 5
Broker's Sending Window (millisec)	1750	2500	4000	6000	12000	18000
Client Sending Window (millisec)	2000	3500	6000	12000	18000	30000

During the testing phase of Juxta-CAT we observed that the number of lost messages ranges in 0% - 1% of the total number of sent messages, which requires sending them again.

4.2 The allocation of jobs to resources

Allocation of jobs to resources is a key issue in order to assure a high performance of applications running on the Juxta-CAT. In this section we show the algorithm implemented by brokers to allocate requests to the resources (see Fig. 11 for an UML diagram representation.)

The algorithm can be seen as a simple price-based model and uses historical information maintained by Juxta-Cat brokers. Therefore, it is fundamental for any broker to maintain updated information about the state of the network and statistics on the performance of the nodes of the grid.

When a broker receives an execution request, it will consult its own historical information and use it (according to the price-model) to determine the price of the allocation to different nodes and finally will decide which is the "cheapest" candidate to execute this request. This policy of task allocation is common for all brokers in Juxta-CAT.

The allocation algorithm works as follows. Based on the historical information on the nodes, the broker uses a set of criteria to compute a score for each candidate node. These criteria are quantitative and can be independent among them. Altogether, these criteria must contribute to bring knowledge to the broker about the "economic saving" that would report any candidate node for the given task resolution. The score for any node is computed according to the following criteria:

1. Total amount of resolved jobs: the larger is this number the better is the node and vice-versa.
2. Number of enqueued jobs waiting for execution: the larger is this number the worse is the node and vice-versa.
3. Number of enqueued JXTA messages waiting to be sent: the larger is this number the worse is the node and vice-versa.
4. Average number of resolved *versus* failed jobs: the larger is the number of successfully resolved jobs and smaller the number of uncompleted jobs, the better is the node and vice-versa.
5. Average number of successfully sent messages to the P2P network: the larger is this number the better is the node and vice-versa.

We use a simple scoring system. The candidate which receives the best score in a criterion will receive a fixed value of 10 points. The second best node has the second better score, 8 points. The rest of scores are 6, 5, 4, 3, 2 and 1 respectively. Furthermore, the scores of the candidates are weighted according to the user's priority, that is, the user of the application can indicate to the Juxta-CAT which is the most important criterion, the second most important criterion and so on.

Thus, by letting N the number of candidate nodes for the task resolution, K the total number of criteria, w_i the weight or priority of criterion i and $S(i,j)$ the score of the ith candidate under criterion j, then the total score S_i of the ith candidate node is:

$$S_i = \sum_{j=1}^{K} w_i \cdot S(i,j).$$

The best candidate is then the one of maximum score, that is, candidate node $i_{\max}$ such that

$$S_{i_{\max}} = \max_{1 \leq i \leq K} S_i.$$

It should be noted that it is very important to keep the historical information updated in the cache of each broker. To achieve this, any node periodically communicates to the brokers its recent statistics. The statistics generated and sent by broker peers are:

- *JXTA statistics* regarding: sent messages, lost messages, current message queue size, average time in message delivery, visible nodes, connections and disconnections to the network.
- *Brokering statistics* regarding: successfully assigned requests, pending requests in the queue, lost requests, average time in allocating tasks to candidate nodes. Also, the historical information of most recent assignments is maintained.

On the other hand, the statistics generated and sent by client peers are:

- *JXTA statistics* regarding: sent messages, lost messages, current message queue size, average time in message delivery, visible nodes, connections and disconnections to the network.
- *Execution statistics* regarding: number of requests accepted, successfully completed tasks, uncompleted tasks, number of pending tasks.
- *Statistics on sent requests* regarding: number of requests sent to the grid, number of requests waiting for the broker response, number of requests waiting for the response of the remote node, number of requests successfully completed, lost or cancelled requests and requests under execution.

In order to publish its statistics, each node generates at a certain interval rate an XML document storing the statistical information indicated above. This document is sent then to the P2P network as an advertisements through the JXTA Discovery Service. This service publishes the advertisement in the local cache of the node and in the cache of the Rendezvous node of the peergroup. After that, it is the Rendezvous node who propagates this information to the rest of caches of the grid nodes.

Although this allocation algorithm is simple, it uses relevant information as regards the performance of the network. This model is flexible as regards the number of criteria to be used as well as the weights to be given to these criteria.

5. EVALUATION OF THE JUXTA-CAT PLATFORM

We have completed a first evaluation of the proposed platform. At this stage of the work, the objective of the evaluation was twofold:

- First, to see the feasibility of the Juxta-CAT as a distributed computing environments regarding scalability, robustness and consistency of the network. In other terms we wanted to evaluate that the network allows broker peers and client peers to join and perform their responsibilities and also end users to submit and solve their tasks using the platform.
- Secondly, to measure the performance of the Juxta-CAT as regards the efficiency of solving problems. We are concerned here mainly with the speedup obtained in solving problems decomposable into independent tasks submitted to the grid. But, also we wanted to measure the time needed by the Juxta-CAT to allocate the tasks to the grid nodes.

We describe next the scenario used for Juxta-CAT testing. It is important to notice that the evaluation process is carried out in a real grid of nodes (see Subsect. 5.2) consisting in a set of geographically distributed real machines.

5.1 Evaluation scenario

We have chosen a simple application scenario yet having rather reasonable computational cost. This is the computation of Π number using approximation series, more precisely the Gregory's series:

$$\Pi = 4 \cdot \sum_{i=0}^{N=\infty} (-i)^i \frac{1}{2i+1}.$$

This problem consists in approximately computing the value of Π as a sum of N fractions, for a given value of N. We run a simple java program on a local machine (P4 2.4 Ghz 512 Mb RAM) that solves the problem and measured the time for different input sizes (values of N) given in Table 2.

Table 2: Sequential execution time of approximating Π.

N	Result	Computation time
10	3,0418396189294032	trivial
10^3	3,1405926538397941	trivial
10^6	3,1415916535897744	trivial
10^9	3,1415926525880504	2 minutes 9 secs
$2 \cdot 10^9$	3,1415926530880768	4 minutes 16 secs
$3 \cdot 10^9$	3,1415926532549256	6 minutes 36 secs

This problem is efficiently parallelized by splitting[1] the whole series into as many parts as nodes of the grid to be used, sending each part to the nodes and sum up the partial results. Thus in this scenario, we generate as many tasks as number of nodes to be used in computation, submit these tasks to the grid and notify the final result. Thus, we had to basically implement the following classes in Java:

- *Sum.java*: receives in input parameters i_{init} and i_{final} and computes the sum of fractions comprised between i_{init} and i_{final}. This class will be run by client peers of the grid.
- *Collector.java*: receives in input a text file each line of which is the partial computed by a peer and computes the final result.

[1]One could use a more efficient version known as the classical partial sum problem.

These classes are then packed in jar files. Thus, for instance, an end user can submit the following four requests[2] to compute the approximate value of Π for $N = 10^9$ terms:

```
java -cp samples.jar juxtacat.samples.pi.Pi -params
0 499999999
java -cp samples.jar juxtacat.samples.pi.Pi -params
500000000 999999999
java -cp samples.jar juxtacat.samples.pi.Pi -params
1000000000 1499999999
java -cp samples.jar juxtacat.samples.pi.Pi -params
1500000000 1999999999
```

Note that the efficiency of approximating Π will certainly depend on the efficient allocation of tasks to the nodes of the grid by the broker peers.

For any instance of the problem, the total execution time (from submitting the task to the grid till outputting the final result) is composed of the time needed to allocate tasks to grid nodes, communication time and the proper computation time of the nodes. Thus, in conducting the experiment we measured the speedup of the computing the approximate value of Π in grid as the number of grid nodes used increases and also the time needed by the grid to allocate the tasks as the number of task increases. Speedup is defined in the usual way:

$$speedup = \frac{T_{seq}}{nbNodes \cdot T_{grid}},$$

where T_{seq} is the sequential resolution time and T_{grid} the resolution time in the grid using $nbNodes$.

5.2 Platform settings

In order to deploy Juxta-CAT in a real grid, we joined the PlanetLab platform [1]. The sample set of PlanetLab's machines prepared for this analysis is about 20 nodes distributed around the European continent. The following nodes[3] were used to install[4] the CMD Juxta-Cat version (client without GUI).

Table 3: Nodes added to the PlanetLab slice.

Host	Description
planet1.manchester.ac.uk	University of Manchester
lsirextpc01.epfl.ch	École Fédérale de Lausanne
planetlab1.polito.it	Politecnico di Torino
planetlab1.info.ucl.ac.be	University of Louivain
planetlab2.upc.es	Universitat Politècnica de Catalunya
planetlab1.sics.se	Swedish Institute of Computer Sci.
planetlab1.ifi.uio.no	University of Oslo
planetlab3.upc.es	Universitat Politècnica de Catalunya
planetlab1.ls.fi.upm.es	Universidad Politécnica de Madrid
planetlab1.hiit.fi	Technology Institute of Helsinki
planetlab-1.cs.ucy.ac.cy	University of Cyprus
planetlab1.ru.is	University of Reykjavik
planetlab2.sics.se	Swedish Institute of Computer Sci.
planetlab1.mini.pw.edu.pl	Telekomunikacja Polska Warsaw
planetlab1.cs.uit.no	University of Tromso
planetlab-02.ipv6.lip6.fr	Laboratoire d'Informatique de Paris

We have also used a small cluster[5] of 6 machines: nozomi.lsi.upc.edu, which manages the processes and has access to Internet (Celeron 2.5 Ghz, 1 Gb RAM, 2 HD IDE RAID1) and 5 equal nodes (AMD64X2 4.4 Ghz, 4 Gb RAM DDR ECC, 2 HD IDE RAID1) making an independent Gigabit network.

5.3 Computational results

Computational results were obtained for the execution scenarios given in Table 4 the experimental setting.

Table 4: Execution scenarios for approximating Π.

N	Local	p hosts ($p = 2, 4, 8, 12, 16$)
10^9	5 tests	5 tests $p \cdot 10$ petitions
$2 \cdot 10^9$	5 tests	5 tests $p \cdot 10$ petitions
$3 \cdot 10^9$	5 tests	5 tests $p \cdot 10$ petitions

For each combination (input size, number of machines) results[6] are averaged over 5 executions; the standard deviation is also presented. We present[7] in Fig. 12 and Fig. 13 the resolution time and speedup for $N = 10^9$ and in Fig. 14 and Fig. 15, for $N = 2 \cdot 10^9$, rspectively.

In these figures we can observe:

- Using grid nodes yields a clear reduction in resolution times. However, increasing the number of grid nodes is "justified" as far as the complexity of the problem needs more processors. In our case, using more than 16 grid nodes does not help in reducing the resolution time. In fact, from Fig. 12 we see that for $N = 10^9$ this "critical" number of processors is around 8 and increases up to 12 when the input size is doubled (see Fig. 14).
- A similar observation holds for speedup. As the number of grid nodes participating in the resolution of the problem increases, the speedup decreases due the increment in the communication times and the time needed by the grid to allocate tasks to grid nodes. Again, by considering a certain number of grid nodes the speedup increases as the input size of the problem increases. Thus, for $N = 10^9$ and 16 grid nodes the speedup is around 0.35 (see Fig. 13) and when the input size is doubled the speedup is around 0.45 (see Fig. 15) and increases up to 0.6 for $N = 3 \cdot 10^9$.

We also measured the time needed by the grid system to allocate the requests, that is, the *brokering time.* We give in Fig. 16 the slowest and averaged brokering time for different numbers of hosts. As can bee seen from this figure, there is a reasonable increase (which seem to be proportional to the increase in the number of grid nodes) in the average brokering time.

[2]In general, just one request is submitted.

[3]PlanetLab node satisfy the following minimum requirements: CPU of 2.0 Mhz, 1024 Mb RAM.

[4]We remark that we have access to PlanetLab nodes only through SSH connections on text-mode.

[5]At the Department of Languages and Informatics Systems, Polytechnic University of Catalonia, Spain

[6]See also the documentation section related to the Juxta-CAT at https://juxtacat.dev.java.net.

[7]We omit the figures for $N = 3 \cdot 10^9$.

Resolution Time (milliseconds)

Variable	Tests	Mean	StDev
Local	5	122363	603
Hosts=2	5	65107	677
Hosts=4	5	35604	809
Hosts=8	5	23554	895
Hosts=12	5	22091	1514
Hosts=16	5	23831	3044

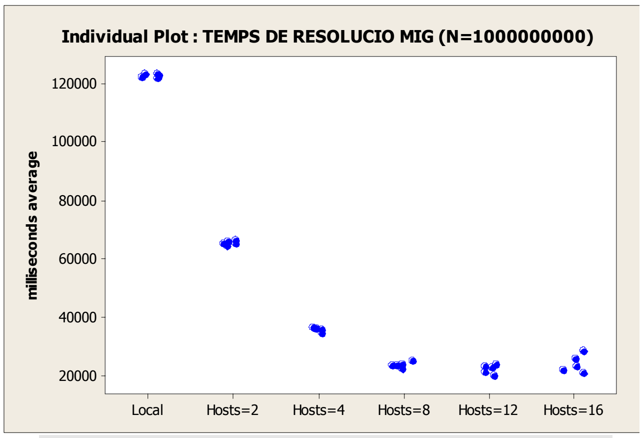


Figure 12: Resolution time for input size $N = 10^9$.

Speed-up

Variable	Tests	Mean	StDev
Hosts=2	5	0,9370	0,0097
Hosts=4	5	0,8570	0,0198
Hosts=8	5	0,6482	0,0248
Hosts=12	5	0,4620	0,0327
Hosts=16	5	0,3240	0,0397

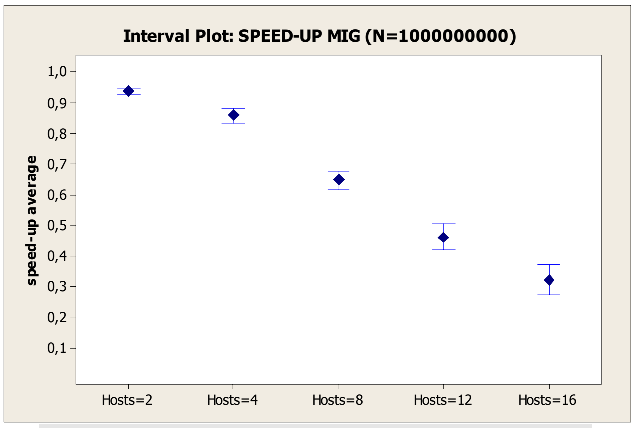


Figure 13: Speedup for input size $N = 10^9$ and different numbers of grid nodes.

Resolution Time (milliseconds)

Variable	Tests	Mean	StDev
Local	5	246648	759
Hosts=2	5	126993	980
Hosts=4	5	66423	817
Hosts=8	5	39828	1535
Hosts=12	5	32649	2156
Hosts=16	5	32324	2576

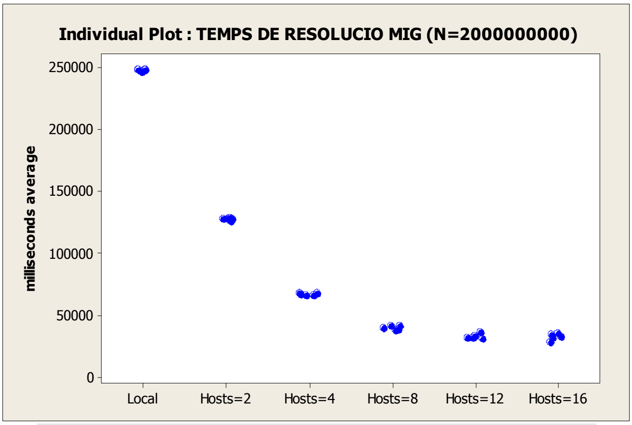


Figure 14: Resolution time for input size $N = 2 \cdot 10^9$.

Speed-up

Variable	Tests	Mean	StDev
Hosts=2	5	0,9686	0,0075
Hosts=4	5	0,9260	0,0114
Hosts=8	5	0,7730	0,0300
Hosts=12	5	0,6330	0,0370
Hosts=16	5	0,4782	0,0400

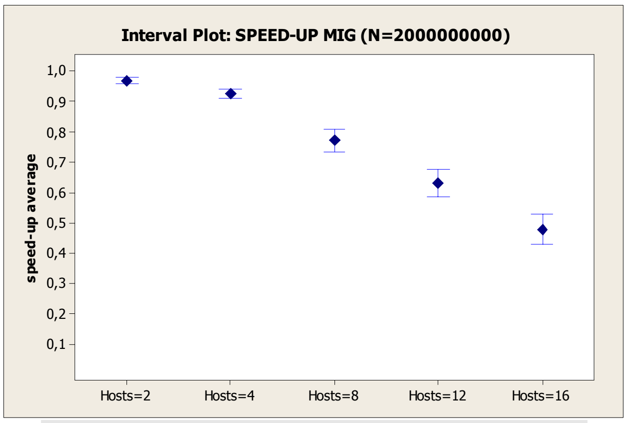


Figure 15: Speedup for input size $N = 2 \cdot 10^9$ and different numbers of grid nodes.

Variable	Tests	Slowest	Average
Hosts=2	15	4025	2188
Hosts=4	15	6868	3985
Hosts=8	15	12555	7851
Hosts=12	15	19788	10858
Hosts=16	15	28455	14655

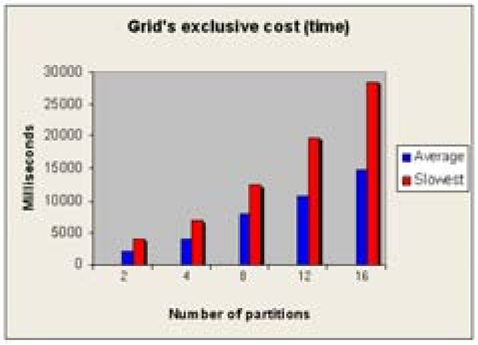


Figure 16: Brokering time.

6. CONCLUSIONS AND FURTHER WORK

In this work we have presented the Juxta-CAT platform developed using JXTA protocols and Java language. Juxta-CAT offers a P2P system for distributed computing. Juxta-CAT can be used from users to submit their tasks in the form of java programs stored on signed jar files and benefit from the large computing power offered by this distributed platform. Users can join Juxta-CAT either as a simple contributor with their machine(s) or as a client peer. The development of our platform required certain improvement/extensions to the original JXTA protocols, especially as regards the Presence Service and Pipe Service protocols. The architecture of Juxta-CAT relays on two types of peers: broker peers that are in charge of managing the allocation of tasks to grid nodes and client peers.

The Juxta-CAT environment has been deployed in a large-scale, distributed and heterogeneous P2P network that we obtained by joining the PlanetLab platform. We have used some simple scenarios to validate and evaluate the proposed platform. The experimental study show the feasibility of our approach and its usefulness in using the Juxta-CAT to speedup task resolution. Moreover, the Juxta-CAT's architecture does not suppose any obstacle for the "gridification" process of task resolution.

We plan to continue the work on Juxta-CAT. On the one hand, we would like to add new functionalities and improvements. Some of these would comprise:

- Allow the resolutions of problems written in languages other than Java; currently, we can only send statements of problems written in Java language.
- Develop a remote launching system for Juxta-CAT's brokers and clients. Due to security restrictions on PlanetLab nodes, we have to launch each peer by connecting first to the remote machine using SSH.
- Provide new and more powerful economic-based models for allocation of tasks to nodes.

On the other hand, we plan to use the current version of Juxta-CAT to develop more realistic applications. Thus we are envisaging the development of an application to process large log-files of daily activity of students at a Virtual University.

Juxta-CAT is included in one of the best showcases of the Java developer's community, where new interested members who would like to participate can be registered.

Finally, there are other issues related to Juxta-CAT such as security issues not mentioned in this paper. The reader is referred to https://juxtacat.dev.java.net for further details.

Acknowledgements

We would like to thank many participants of the Java.net and Jxta.org for their help. In particular, we are grateful to Daniel Brookshier for his support during the development of Juxta-CAT project.

7. REFERENCES

[1] D. Bickson. Planetlab project how-to. DANSS LAB (Distributed Algorithms Networking and Secure Systems Group).
[2] D. Brookshier, D. Govoni, N. Krishnan, and J.C Soto. *JXTA: Java P2P Programming*. Sams Pub., 2002.
[3] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.
[4] H. Casanova and J. Dongarra. Netsolve: Network enabled solvers. *IEEE Computational Science and Engineering*, 5(3):57–67, 1998.
[5] S.H.. Clearwater, editor. *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific Press, Singapore, 1996.
[6] I. Foster. What is the grid? A three point checklist. White Paper, July 2002.
[7] I. Foster and C. Kesselman. *The Grid-Blueprint for a New Computing Infrastructure*. Morgan Kauf., 1998.
[8] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *International Journal of Supercomputer Applications*, 15(3), 2001.
[9] L. Linderoth and S.J. Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24:207–250, 2003.
[10] S. Oaks, B. Traversat, and L. Gong. *JXTA in a Nutshell*. O'Reilly, 2003.
[11] A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 1st edition, 2001.
[12] T.T. Ping, G.Ch.. Sodhy, Ch.H.. Yong, F. Haron, and R. Buyya. A market-based scheduler for jxta-based peer-to-peer computing system. In *ICCSA (4)*, pages 147–157, 2004.
[13] C. Shikey. What is P2P ... and what isn't. O'Reilly Network, November 2000.
[14] R. Wolski, J. Brevik, J. Plank, and T. Bryan. Grid resource allocation and control using computational economies. In G.Foxand F.Berman and A.Hey, editors, *Grid Computing- Making the Global. Infrastructure a Reality*, chapter 32. Wiley, 2003.
[15] S.J. Wright. Solving optimization problems on computational grids. *Optima*, 65, 2001.

Session D
Resource and Object Management

The Management of Users, Roles, and Permissions in JDOSecure

Matthias Merz
Department of Information Systems III
University of Mannheim
L 5,5, D-68131 Mannheim, Germany
merz@uni-mannheim.de

ABSTRACT

The Java Data Objects (JDO) specification proposes a transparent and database-independent persistence abstraction layer for Java. Since JDO is designed as a lightweight persistence approach, it does not provide any authentication or authorization capabilities in order to restrict user access to persistent objects. The novel security approach, JDOSecure, introduces a role-based permission system to the JDO persistence layer, which is based on the Java Authentication and Authorization Service (JAAS). However, using JAAS policy files to define appropriate permissions becomes more complex and, therefore, error-prone with an increasing number of different users and roles. Thus, JDOSecure comprises a management solution for users, roles, and permissions. It allows storing the information which is necessary for authentication and authorization in any arbitrary JDO resource. Furthermore, a Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

1. INTRODUCTION

The Java Data Objects (JDO) specification proposes a transparent and database-independent persistence abstraction layer for Java [6, 7]. It enables application developers to deal with persistent objects in a transparent fashion. Furthermore, JDO as a data store independent abstraction layer enables the mapping of domain object architectures to any arbitrary type of data store.

JDOSecure [9, 10] is a novel security architecture developed as an add-on for the Java Data Objects specification (more information about JDOSecure could be found at `projekt-jdo.uni-mannheim.de/JDOSecure`). Its objective is preventing unauthorized access to the data store while using the JDO API. It introduces a fine grained access control mechanism to the JDO persistence layer and allows the definition of role-based permissions. In detail, permissions could be set for individual user-roles, a specific package or class, and a *CRUD*[1]-operation. Based on the dynamic proxy approach, JDOSecure is able to collaborate with any JDO implementation without source code modification or recompilation.

In order to authenticate and authorize users accessing a JDO resource, JDOSecure implements the *Java Authentication and Authorization Service* (*JAAS*). The default JAAS implementation uses permissions and simple policy-files to allow or disallow access to crucial system resources. However, defining appropriate permissions in simple text-files becomes more and more complex with an increasing number of different users and roles. In order to reduce possible inconsistency and potential typos, JDOSecure comprises a management solution for users, roles, and permissions. It allows storing the authentication and authorization information in any arbitrary JDO resource. Furthermore, a Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions.

In this paper, we shall focus on the management of users, roles, and permissions in JDOSecure. The outline of the paper is organized as following: Section two introduces the current JDO specification and recalls the design of the Java Authentication and Authorization Service. Section three outlines the architecture of JDOSecure. In section four we will present the new management solution for users, roles, and permissions. The last section will give a critical review and will address topics for future research.

2. TECHNOLOGY OVERVIEW

In this section, we will provide a basic technology overview to the Java Data Objects specification and to the Java Authentication and Authorization Service.

2.1 The Java Data Objects Specification

The Java Data Objects specification is an industry standard for object persistence developed by an initiative of Sun Microsystems under the auspices of the Java Community Process [6]. JDO was introduced in April 2002 and is intended for the usage within the Java 2 Standard (J2SE) and

[1]CRUD is an acronym for Create, Retrieve, Update, and Delete. Persistence Frameworks usually provide appropriate methods to manage persistent objects.

Enterprise Edition (J2EE). It enables application developers to deal with persistent objects in a transparent fashion. Thus, JDO as a data store independent abstraction layer enables the mapping of domain object architectures to any type of data store. The recent version of JDO (JDO 2.0) was finally approved in may 2006 [7]. Its objective is to simplify the usage of JDO e.g. by providing a standardized object/relational mapping format to allow a higher degree of application portability. It also introduces an attach/detach mechanism for persistent objects to facilitate middle tier architectures. One of its most beneficial features is the extension of the JDO Query language (JDOQL) to support e.g projections, aggregates, or named queries.

The JDO specification defines two packages: The JDO Application Programming Interface (API) allows application developers to access and manage persistent objects. The classes and interfaces of the Service Providers Interface (SPI) are intended to be used exclusively by a JDO implementation.

The interfaces and classes of the JDO API are located in the package `javax.jdo` [6, 7]. The `Transaction` interface provides methods for initiation and management of transactions under user control. The `Query` interface allows obtaining persistent instances from the data store by providing a Java-oriented query language called JDO Query Language (JDOQL). The `PersistenceManager` serves as primary application interface and provides methods to control the life cycle of persistent objects. An instance implementing this interface could be constructed by calling the `getPersistenceManager()` method of a `PersistenceManagerFactory` instance. Since `PersistenceManagerFactory` itself is just another interface, constructing an instance prior to this type becomes necessary. It usually could be constructed by calling the static `JDOHelper` method `getPersistenceManagerFactory(Properties props)`. The class `JDOHelper` is also part of the JDO API and enables an easy replacement of the currently preferred JDO implementation without source code modifications in context of an application. The information about the currently used JDO implementation and data store specific parameters has to be passed to this method by a `Properties` object. A user identification and a password in order to access the underlying data store are also part of the `Properties` object. In order to prevent misunderstandings, the JDO persistence approach does not distinguish between different user identifications or individual permissions. With the construction of a `PersistenceManager` instance, the connection to the data store will be established and users are able to access the resource without further restrictions.

Every instance that should be managed by a JDO implementation has to implement the `PersistenceCapable` interface. As part of the JDO SPI package, the `PersistenceCapable` interface does not have to be implemented explicitly by an application developer. Instead, the JDO specification prefers a post-processor tool (*JDO-Enhancer*) that automatically implements the `PersistenceCapable` interface. It transforms regular Java classes into persistent classes by adding the code to handle persistence. An XML-based *persistence descriptor* has to be configured previously. The JDO-Enhancer evaluates this information and modifies the Java bytecode of these classes adequately. The JDO specification assures the compatibility of the generated bytecode for the use within different JDO implementations. The `StateManager` interface as part of the JDO SPI provides the management of persistent fields and controls the object lifecycle of persistent instances.

Although JDO provides a standardized, transparent and data store independent persistence solution including tremendous benefits to Java application developers, the JDO specification has also been critized in the Java community. Besides technical details like the JDO enhancement process [15], the substantial overlaps between the Enterprise JavaBeans specification [5] and JDO [8], as well as the conceptual design as a lightweight persistence approach has been criticized. Some experts even argue that shifting JDO to a more comprehensive approach including distributed access functions and multi-address-space communication [14] is necessary. As a result of it's lightweight nature, JDO does not provide a role-based security architecture, e.g. to restrict the access of individual users to the data store. Consequently, the JDO persistence layer does not provide any methods for user authentication or authorization. Every user has full access privileges to store, query, update and delete persistent objects without further restrictions. For example, using the `getObjectById()` method allows him to receive any persistent object whereas the `deletePersistent()` method enables a user to delete objects from the data store.

At first glance, a slight improvement could be achieved by setting up individual user identifications at the level of the data store. This would allow the construction of different and user dependent `PersistenceManagerFactory` instances. If, however, all users should have access to a common database, individual user identifications and appropriate permissions have to be defined inside the data store. However, configuring user permissions to restrict the access to certain objects is quite complex. For example, when using a relational database management system, the permissions would have to be configured, based on the object-relational mapping scheme and the structure of the database tables. Thus, it leads to the disadvantage of causing a strong dependency between the user application and the specific data store. In addition to that, a later replacement of the data store preferred currently leads to a time-consuming and expensive migration. It is obvious that the strong binding of security permissions to a specific data store contradicts the intention of JDO, which is providing application programmers a data-store-independent persistence abstraction layer.

As JDOSecure is based on the Java Authentication and Authorization Service, the following section will give a brief overview to this approach.

2.2 The Java Authentication and Authorization Service

The Java security architecture is based on three components: Bytecode verifier, class loader and security manager (cf. [13] and [4]). The bytecode verifier checks the correctness of the bytecode and prevents stack overflows. The class loader locates and loads classes into the Java Virtual Machine (JVM) and defines a unique namespace for each class. The security manager or, more accurately, the `AccessController` instance checks the invocation of operations relevant to security e.g. local file system access, the setup of system properties or the use of network sockets (cf. figure 1).

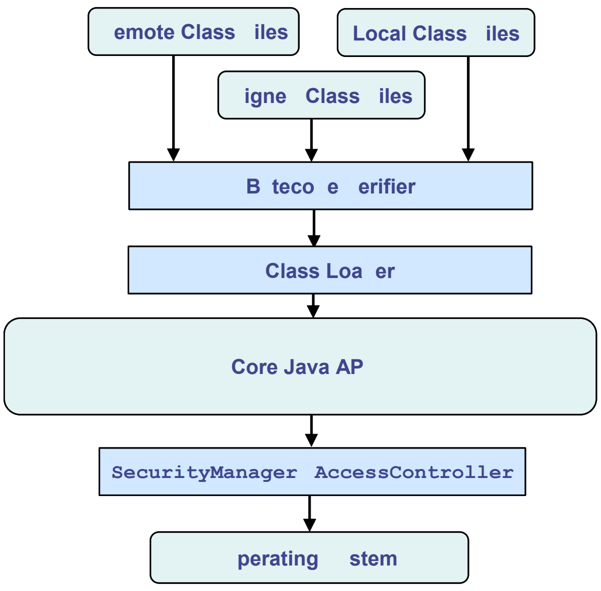


Figure 1: Java 2 Security Architecture, according to [11]

The *code-centric authorization* approach in Java allows to restrict the access to system resources depending on the source and the author/signer of the bytecode. It was intended to ensure that a malicious Java program could not damage the user's system. With the introduction of the Java Authentication and Authorization Service (JAAS) in Java 1.4 it is also possible to restrict the access to resources depending on the currently authenticated user (*user-centric authorization*). During the authentication process, a user is identified by the system e.g. with the help of user identification and password. In Java, this mechanism is implemented by the `SecurityManager` which delegates access-requests to the `AccessController`. This instance validates the permissions and allows or disallows the access to the resources.

Figure 2 and 3 outlines the authentication and authorization process for JAAS. Starting with the authentication process, the application first creates a `LoginContext` and invokes the `login()` method of this instance. As defined in the login configuration file, the `LoginContext` delegates the authentication to one or several `LoginModules`. A `LoginModule` could use a `CallbackHandler` for communication with the application, the environment or the user e.g. to prompt for a password. JAAS implements the PAM (Pluggable Authentication Modules) standard to integrate further authentication services, like Kerberos, Radius or LDAP. If an authentication attempt finally fails, a `SecurityException` will be thrown.

By calling the `getSubject()` method of the `LoginContext`, referring to the `Subject` instance becomes possible. This represents an authenticated user that is associated with one or several `Principals` (a concrete user role). An application can perform a `PrivilegedAction` considering individual user permissions by invoking the static `Subject.doAs(subject, action)` method. Therefore, the `AccessController` determines if a `Subject` is associated with at least one `Principal`, that provides the permissions

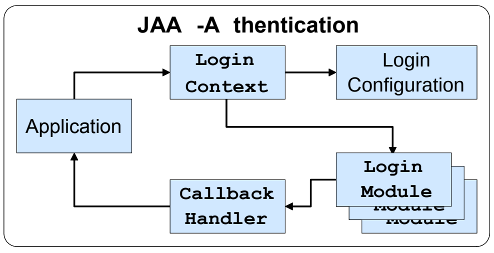


Figure 2: JAAS-Authentication

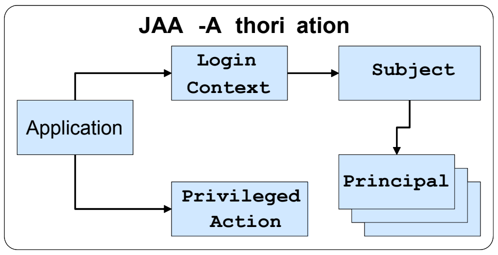


Figure 3: JAAS-Authorization

necessary to perform this action. If a user or application does not have the permission necessary to perform this `PrivilegedAction`, a `SecurityException` will be thrown. The relationship between permissions on the one hand and `Principals` on the other is defined in a separate *policy-file* (cf. section 3.4).

3. THE ARCHITECTURE OF JDOSECURE

This section outlines the system architecture of JDOSecure. The basic authentication and authorization concepts will be introduced and subsequently, the integration of JDOSecure with a JDO implementation is covered.

3.1 JDOSecure Preface

As already noted, JDOSecure will introduce a role-based permission system to the JDO persistence layer based on the Java Authentication and Authorization Service.

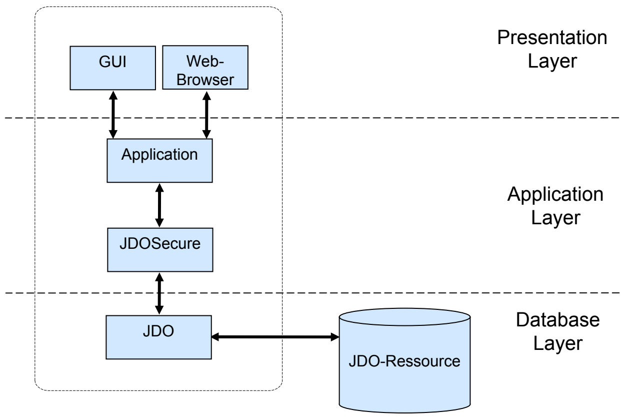


Figure 4: Interposition of JDOSecure between an Application and a JDO Implementation

As pictured in Figure 4, JDOSecure is intended to be interposed between an application and a JDO implementation. Thus, the architecture of the application domain could consist of user-interface, application, JDOSecure, JDO implementation and a JDO resource to store the persistent objects of the application domain. If e.g. a user was connected to a Java application by using a graphical user interface (GUI) or a web browser, the user will not even be aware of the interception.

3.2 The Authentication Process

As described in section 2.1, a `PersistenceManager-Factory` instance can be invoked by calling the static `getPersistenceManagerFactory(Properties props)` method of the `JDOHelper` class. JDOSecure extends this concept in order to facilitate the collaboration between JDOSecure and any JDO implementation. Hence, JDOSecure provides a `JDOSecureHelper` class which is derived from `JDOHelper`. The `JDOSecureHelper` class overrides the `getPersistenceManagerFactory(Properties props)` method and serves as an entry-point for JDO applications.

The `Properties` object passed to the `JDOHelper` class contains amongst others user identification and password to access a JDO resource. As mentioned in section 2.1 the JDO architecture does not distinguish between different users. Therefore, the `JDOSecureHelper` analyzes the passed `Properties` object to authenticate a user at the level of the JDO persistence layer. Once a user has authenticated successfully, the `JDOSecureHelper` class constructs a new `PersistenceManagerFactory` instance. The basic idea in this context is to replace username and password in the `Properties` object, before the `JDOSecureHelper` class invokes the `getPersistenceManagerFactory(Properties props)` method of the original `JDOHelper` class. The intention of this replacement is to prevent a direct connection between user and JDO resource by using the `JDOHelper` class instead of the `JDOSecureHelper` class as a ”workaround”. The replaced password is unknown to the user and has to be configured by a security-administrator for the JDOSecure implementation and the JDO resource previously.

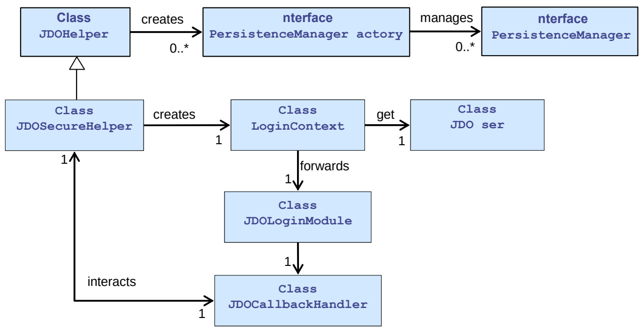


Figure 5: Using JAAS to Implement User Authentication in JDOSecure

As illustrated in Figure 5, a `LoginContext` instance will be constructed by invoking the `getPersistenceManagerFactory()` method of the `JDOSecureHelper` class. The `LoginContext` instance forwards the authentication-request to the `JDOLoginModule` and `JDOCallbackHandler`. The `JDOCallbackHandler` instance validates the `ConnectionUserName` and the `ConnectionPassword` property to authenticate the user. If this process completes without throwing a `SecurityException` the `LoginContext` instance is associated with a `JDOUser` instance and a `PersistenceManagerFactory` instance is constructed. As described in the next section, the `JDOSecureHelper` instance does not return a `PersistenceManagerFactory` instance, but a proxy instance to implement the access control mechanism instead.

3.3 JDOSecure and the Dynamic Proxy Approach

There are two prerequisite conditions that could affect the acceptance of JDOSecure. First, JDOSecure should be independent from a concrete JDO implementation to ensure an ongoing portability between different JDO implementations. And secondly, an overall approach should not contradict the JDO specification. In an attempt to meet these requirements, the presented security architecture implements the *dynamic proxy* pattern [1]. As it will be described in the following, this concept enables the collaboration between JDOSecure and a standard JDO implementation, without an extensive adaptation.

A proxy instance implements the interfaces of a specific object and allows one to control access to it [3]. Generally, the creation of a proxy has to be done at compile time. Moreover, the dynamic proxy concept allows the dynamic construction of a proxy instance at runtime [1]. Dynamic proxy instances are always associated with an `InvocationHandler` and could be created e.g. by using the static `newProxyInstance()` method of the `java.lang.reflect.Proxy` class. Any method invocation directed to proxy instance will be redirected to the `InvocationHandler.invoke()` method. The `invoke()` method allows to intercept method calls before they are forwarded to the original object.

The JDOSecure architecture implements the dynamic proxy concept as shown in Figure 6. The basic idea is to interpose a proxy between `PersistenceManager` and a JDO user or application. This would allow to validate specific user permissions at the `PMInvocationHandler` instance, before a method call is forwarded to the `PersistenceManager`. The following paragraph will explain the architecture more in detail.

As mentioned above, the `JDOSecureHelper.get-PersistenceManagerFactory()` method returns a dynamic proxy instance of the `PersistenceManagerFactory` class. Thus, the JDOSecure architecture avoids a direct interaction with the original `PersistenceManagerFactory`-instance and allows to manipulate method calls which are directed to the `PersistenceManagerFactory`. Invoking the `getPersistenceManager()` method, the `PMFInvocationHandler` returns a second proxy, in this case a proxy of the `PersistenceManager` instance. JDOSecure uses the associated `InvocationHandler` (`PMInvocationHandler`) to manipulate method calls directed to the `PersistenceManager`. Thus, the `PMInvocationHandler` represents the entry-point in order to implement the authorization function and allows one to determine whether or not a user is allowed to invoke a `PersistenceManager` method.

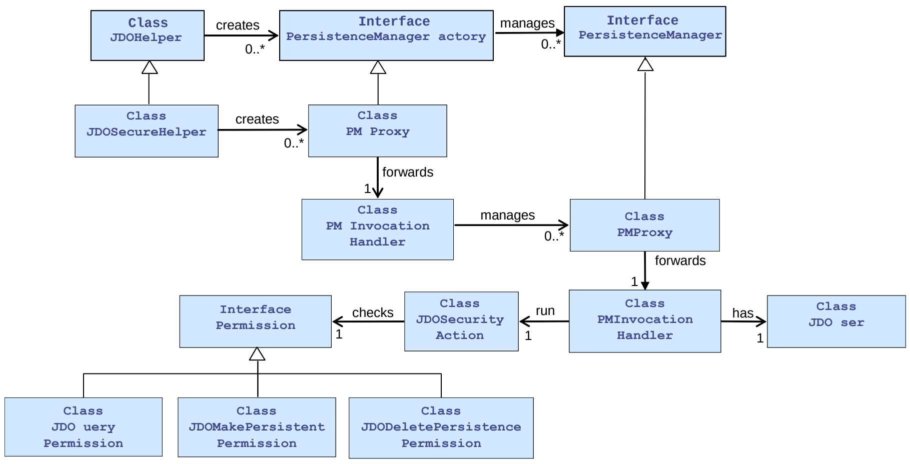

Class JDOHelper
creates
0..*
Interface PersistenceManager actory
manages
0..*
Interface PersistenceManager
Class JDOSecureHelper
creates
0..*
Class PM Proxy
forwards
1
Class PM Invocation Handler
manages
0..*
Class PMProxy
forwards
1
Interface Permission
checks
1
Class JDOSecurity Action
run
1
Class PMInvocation Handler
has
1
Class JDO ser
Class JDO uery Permission
Class JDOMakePersistent Permission
Class JDODeletePersistence Permission

Figure 6: Using the Dynamic Proxy Approach to Implement User Authorization

3.4 The Authorization Process

JDOSecure enables the set-up of user specific permissions in order to allow or disallow the invocation of `PersistenceManager` methods. As already mentioned, a user receives a proxy of a `PersistenceManager` instance (`PMProxy`) by invoking the `getPersistenceManager()` method. Thus, JDOSecure is able to use the assigned `PMInvocationHandler` to validate, if an authenticated `JDOUser` has the permission to make a specific method invocation. The permissions are located in a separate policy-file and can be individually defined for any user. Currently, JDOSecure distinguishes between different permissions (Table 1) in order to restrict the access to the different `PersistenceManager` methods. JDOSecure also enables the limitation of user permissions to a certain package or a specific class.

For instance, the permission to invoke the `makePersistent()` method could be defined for a package `org.test.sample` and a single user "sampleuser" as following:

```
grant Principal JDOUser "sampleuser"{
 permission JDOMakePersistentPermission
                  "org.test.sample.*";
}
```

In order to validate if a user has the permission to invoke a specific `PersistenceManager` method, a `JDOSecurityAction` instance will be constructed and passed to the static `doAs(subject, action)` method of the `Subject` class. Consequently, the validation of a user permission is delegated to the `AccessController` as part of the Java 2 Security Architecture. If a user has the appropriate permission to invoke a specific `PersistenceManager` method, the method call is forwarded to the original `PersistenceManager` instance. If not, a Java `SecurityException` is thrown and the access to the JDO resource is rejected.

Even this approach allows one to restrict the creation, query and deletion of `PersistentCapable` instances, it is not suitable for the JDO update process. This problem is addressed in the next section.

3.5 JDOSecure and the Update of Object Attributes

JDO introduces the concept of transparent persistence and consequently JDO doesn't provide any additional methods to update object attributes or flushing instances to the data store. The security mechanism as described above, to verify user permissions when invoking methods of the JDO API, does not work in case of JDO updates.

As already mentioned, the JDO enhancer modifies regular Java classes in order to implement the `PersistentCapable` interface. Additionally, all setter methods are modified, that they do not change attributes directly. Instead, by invoking a setter method, an associated `StateManager` instance will be notified. This `StateManager` is responsible to update the attributes in the corresponding `PersistentCapable` instance as well as to propagate these updates to the database.

The idea in this context is to replace the `StateManager` by another proxy and to validate the user permissions in the corresponding `InvocationHandler` instance. As defined in the JDO specification, a `StateManager` instance will be created by the JDO implementation with the invocation of the `PersistenceManager` methods `makePersistent(...)`, `makePersistentAll(...)`, `getExtent(...)`, `getObjectById(...)` as well as the `execute(...)` method of the `Query` instance. With the use of JDOSecure, the user does not interact with the `PersistenceManager` directly, but with the `PMInvocationHandler` instance. Before JDOSecure returns a `PersistentCapable` instance to the user, replacing the corresponding `StateManager` by a proxy becomes possible.

In order to implement this approach in JDOSecure, the `PMInvocationHandler` accesses the private `jdoStateManager` field by using the `java.lang.reflection` API to construct a dynamic proxy for the `StateManager`. In a second step, the `PMInvocationHandler` replaces the reference to the `StateManager` in the `PersistentCapable` instance with the proxy. The technical details like security issues when accessing private fields by using the `java.lang.reflection` API and other complications (e.g. the `jdoReplaceStateManager(...)` method of a `StateManager`) have been disregarded in order to improve

Methods of a `PersistenceManager`, that require specific permissions to be executed in the context of JDOSecure:	Necessary permission to invoke the according method for a specific class or package:
`makePersistent(..)` `makePersistentAll(..)`	`JDOMakePersistentPermission` $< Class >$
`deletePersistent(..)` `deletePersistentAll(..)`	`JDODeletePersistentPermission` $< Class >$
`getExtent(..)` `Query.execute(..)`	`JDOQueryPermission` $< Class >$
-	`JDOUpdatePermission` $< Class >$

Table 1: JDOSecure Permissions

clarity. However, JDOSecure enables the access control of the JDO update mechanism by introducing another proxy and a `JDOUpdatePermission`. As all other JDOSecure permissions, the `JDOUpdatePermission` could be specified individually for every user and a specific package or class.

4. THE MANAGEMENT OF USERS, ROLES, AND PERMISSIONS IN JDOSECURE

This section focusses on the management of users, roles, and permissions in JDOSecure. At first, the interaction between the users, roles, and permissions management system and JDOSecure will be described. Subsequently, the domain model of the users, roles and permissions management system will be outlined and the application to maintain the appropriate information will be introduced.

4.1 Interaction between the Users, Roles, and Permissions Management System and JDOSecure

The JDOSecure users, roles, and permissions management system allows to store the information which is necessary for authentication and authorization in a separate JDO resource. The interaction between the management component and JDOSecure on level of the application layer is illustrated in Figure 7. The left part of this illustration corresponds to Figure 4. The right part describes the schematic architecture of the users, roles, and permissions management system consisting of the administration utility, a users, roles, and permission management component, a JDO-Implementation and a separated JDO resource in order to store the administration-data.

The UML Component diagram in Figure 8 represents the interaction more in detail. Since JDOSecure uses JAAS, the abstract Java class `Policy` and the JAAS interface `LoginModule` are used as interfaces. The `JDOLoginModule` class implements the interface `javax.security.auth.spi.LoginModule` and the class `JDOPolicy` extends the abstract class `java.security.Policy`. The class `JDOLoginModule` implements the PAM standard and enables the access to the necessary data like user name and password for the authentication process. In order to enable JAAS to use the `JDOLoginModule` to authenticate users at runtime, the JAAS Login Configuration file has to be adapted previously [12]. The instance of `LoginContext`, which will be constructed by the `JDOSecureHelper`, analyzes the configuration file

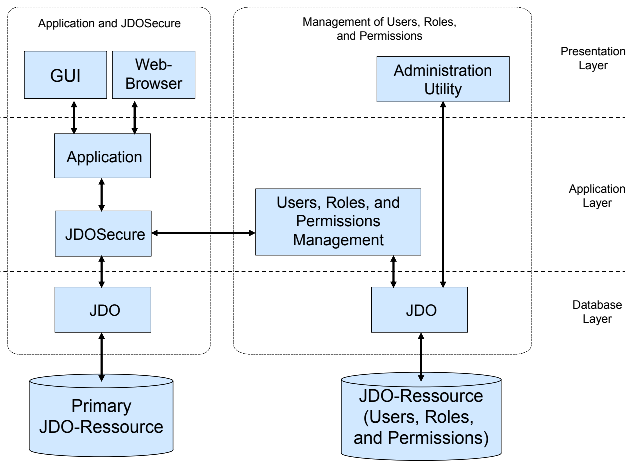

Application and JDOSecure
Management of Users, Roles, and Permissions
Presentation Layer
GUI
Web-Browser
Administration Utility
Application
Application Layer
Users, Roles, and Permissions Management
JDOSecure
Database Layer
JDO
JDO
Primary JDO-Ressource
JDO-Ressource (Users, Roles, and Permissions)

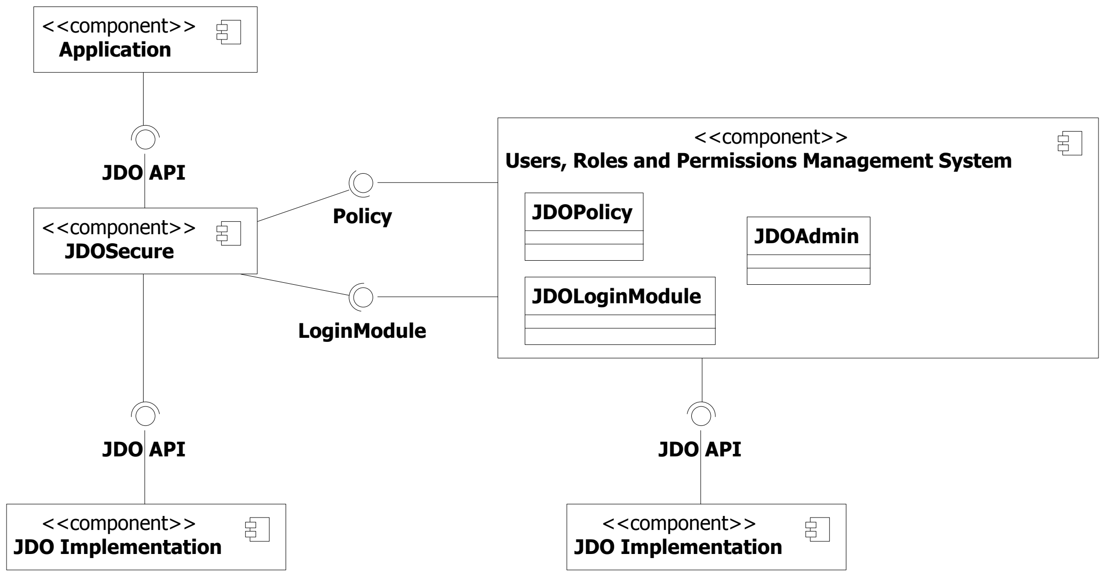
Figure 7: Binding between the Users, Roles, and Permissions Management System and JDOSecure

and determines, which Login-Modul has to be used for the authentication process.

As mentioned in Figure 5, the `JDOCallbackHandler` passes the parameters user identification and password to the `JDOLoginModule`. If a `Credential` object in the JDO resource matches the same user identification and password, the authentication process completed successful. In this case the `JDOLoginModule` passes all `Principal` objects that are associated with the `Credential` object to the `LoginContext`. Finally, JAAS adds these `Principal` objects to the current session (`Subject`). The class `JDOPolicy` extends the default Java `Policy` implementation and grants access to the data for authorizing the users (`Principal` objects, permissions). In order to enable the `JDOPolicy` class to perform the authorization process, an instance of this class has to be constructed and activated for the entire JVM at run-time.

In order to manage the authorization process, the `JDOPolicy` instance detects the permissions of all `Principal` objects assigned to the current session as well as the permissions granted to a `CodeSource`[2] object. In order to achieve data store independency, the permissions are mapped into a JDO resource and could be directly accessed from the

[2]A `CodeSource` object encapsulates a code-base (`java.net.URL`) and public key certificates of the classes being loaded.



<<component>> Application
JDO API
<<component>> JDOSecure
Policy
LoginModule
<<component>> Users, Roles and Permissions Management System
JDOPolicy
JDOAdmin
JDOLoginModule
JDO API
JDO API
<<component>> JDO Implementation
<<component>> JDO Implementation

Figure 8: UML Component Diagram Identifying the Components with regard to JDOSecure

`JDOPolicy` instance. The `JDOAdmin` class represents an administration utility including a graphical user interface that will simplify the management of users, roles and permissions for JDOSecure. More details will be discussed in section 4.3. The underlying domain model of the users, roles, and permissions management system will be presented in the following section.

4.2 Modelling the Users, Roles, and Permissions Management Domain

The domain model of the users, roles, and permissions management system was designed to be as flexible as possible. Therefore, the classes `Principal`, `CodeSource` and `Permission` are widely independent from the methods to mange the corresponding objects. This allows to add further management methods in the Java-based administration utility without any changes to the domain model.

In Figure 9, the domain model is represented. The authentication and the authorization part are highlighted each by a dotted line. The linking element between this two parts is the class `Principal`. Since a user can be identified in several ways, the class `Credential` represents the information that is required to authenticate a user e.g. by using a user identification and password. The user himself is not explicitly represented in JAAS. However, the user is implicitly represented by the references between the `Credential` and the `Principal` objects.

The set of allowed actions is concatenated by an instance of the `AuthorizationEntry` class. In detail, the permissions of a specific user could easily be determined by aggregating all permissions that are assigned to all user's identities and the assigned `CodeSource` instances. The `CodeSource` class enables an administrator to bind permissions to a URL, which defines from where a Java class is being loaded. The next section will introduce the Java-based administration utility to mange users, roles and permissions.

4.3 Administration Utility to Mange Users, Roles, and Permissions

The Java-based administration utility with the function of maintaining users, roles, and permissions is introduced to simplify the management of the information which is necessary for authentication and authorization. It is designed to be used by a security administrator, even from a remote computer by using a direct JDO connection. In order to provide a very comfortable solution, the graphical user interface possess two different management modes:

The first management mode enables an administrator to operate directly on the presented domain model. Thus, the `AuthorizationEntry` objects could be directly constructed and bound to a `Principal`, `Permission` and `CodeSource` objects. Even this mode is highly effective in granting permission to a specific `CodeSource` and a `Principal`, it also has the disadvantage of a very complex handling.

Therefore, a second management mode is geared to the *Role-Based Access Control* (*RBAC*) standard [2]. In the so-called RBAC mode, the permission management of `Principals` is separated from the permission management of `CodeSource` objects. Thus, the RBAC mode has the advantage of an easier handling, but on the other hand, it is e.g. not possible to grant permissions to set of of a single user/role and a specific `CodeSource` object. Also permissions can only be defined for a single user role (not for collection of user roles), which could lead to the definition of new roles. Figure 10 outlines the assignment of permissions and roles using the RBAC management mode. Moreover, as can be seen, the Java-based administration utility also allows to define Java permissions, e.g like a Java `FilePermission`. Even these permissions are not directly related to JDO; they are generally required, due to the activated `SecurityManager`.

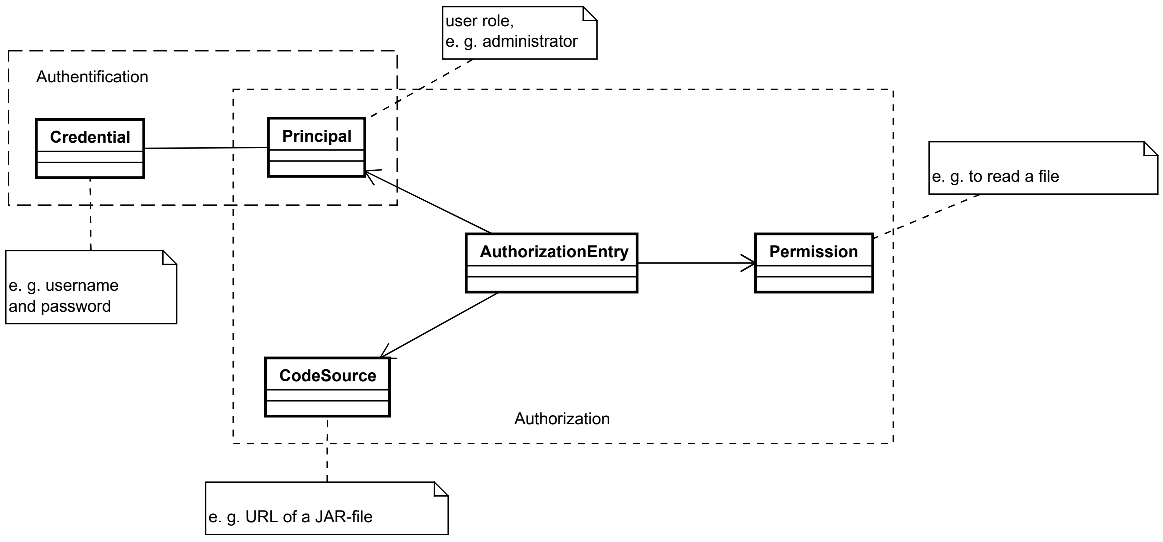


Figure 9: UML Class Model of the Users, Roles, and Permissions Management Domain

5. CONCLUSION

In this article, the JDOSecure architecture is introduced and the main advantages are highlighted. JDOSecure introduces a fine-grained access control mechanism to the JDO persistence layer and allows the definition of role-based permissions. The permissions could be defined individually for every user/role with regards to certain operations (create, delete, update, and query) and a specific class/package. In order to authenticate and authorize users accessing a JDO resource, JDOSecure implements the Java Authentication and Authorization Service. As it turns out, defining appropriate permissions using JAAS policy files becomes more complex and error-prone with an increasing number of different users and roles. Therefore, JDOSecure comprises a management solution for users, roles, and permissions. This approach allows to map the information which is necessary for authentication and authorization in any arbitrary JDO resource. A Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions. Based on the dynamic proxy approach, JDOSecure is able to collaborate with any JDO implementation without source code modification or recompilation.

Even JDOSecure could improve the security of JDO applications, one potential shortcoming of JDOSecure should also be mentioned. Since JDOSecure provides a fine-grained access control mechanism, it becomes obvious that the management of permissions and the access control mechanism has negative performance affects. Even worse, the dynamic proxy approach including a huge number of indirections between the constructed instances and their proxies leads to a further deterioration of performance. In order to get a first impression of the performance behavior, we have suggested a test-scenario that covers the measurement of CRUD operations. Each test case has been executed several times with a set of different number of persistent objects. It turns out that JDOSecure reduces the performance in this test scenario at about 15-20% on an average. In the future, we aim to extend our performance tests and are confident of achieving a slightly better performance for JDOSecure.

Acknowledgments

I thankfully acknowledge the fruitful discussions with my colleagues from the department of Information Systems III. Especially, I would like to thank Prof. Dr. Martin Schader and Dr. Markus Aleksy for reading this paper thoroughly and for their helpful comments and advice regarding its contents. I appreciate and also thank my colleagues from the University of Mannheim IT Center for their kind and continuous support.

6. REFERENCES

[1] J. Blosser. Explore the Dynamic Proxy API. http://java.sun.com/developer/technicalArticles/DataTypes/proxy/, 2000.

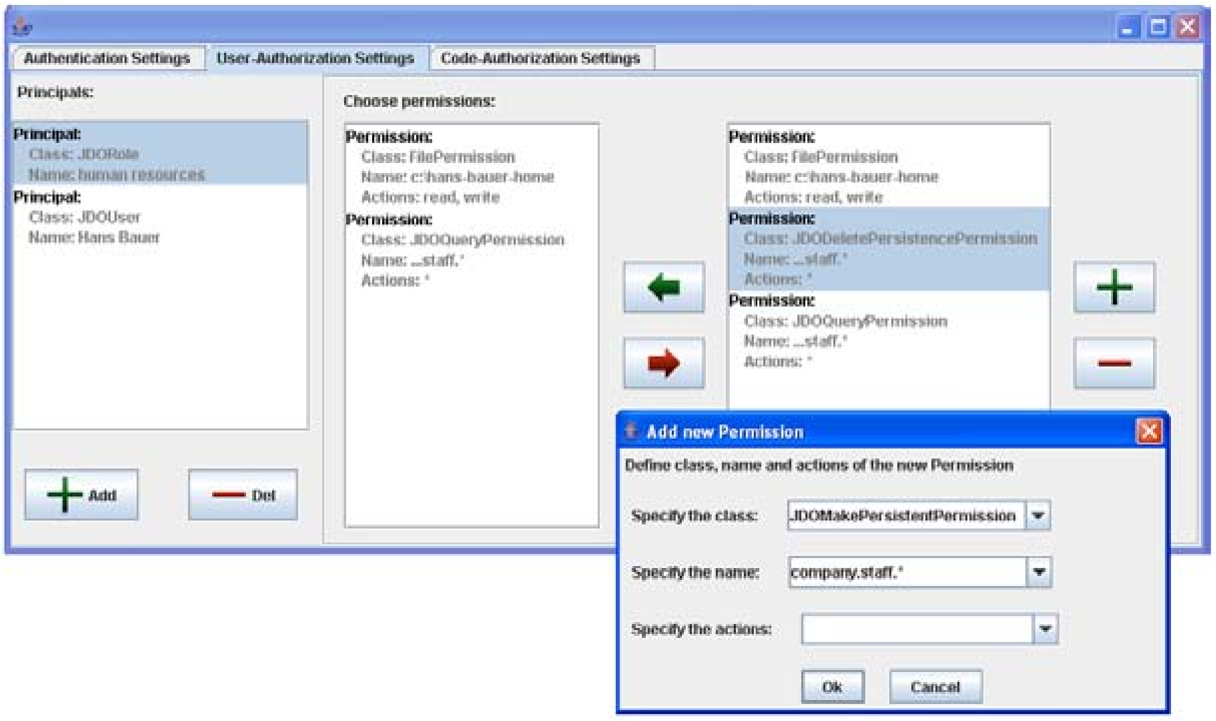
[2] Ferraiolo, David F., Sandhu, Ravi, Gavrila, Serban, Kuhn, D. Richard, and Chandramouli, Ramaswamy. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, vol. 4 no. 3:p. 224–274, 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[4] L. Gong. Java 2 Platform Security Architecture, http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html, 2002.

[5] Java Community Process. JSR-153: Enterprise JavaBeans, Final Release, Version 2.1, http://www.jcp.org/en/jsr/detail?id=153, 2003.

[6] Java Community Process. JSR-012: Java Data Objects (JDO) Specification, Maintenance Release,



Principals:

Principal:
Class: JDORole
Name: human resources
Principal:
Class: JDOUser
Name: Hans Bauer

Choose permissions:

Permission:
Class: FilePermission
Name: c:\hans-bauer-home
Actions: read, write
Permission:
Class: JDOQueryPermission
Name: ...staff.*
Actions: *

Permission:
Class: FilePermission
Name: c:\hans-bauer-home
Actions: read, write
Permission:
Class: JDODeletePersistencePermission
Name: ...staff.*
Actions: *
Permission:
Class: JDOQueryPermission
Name: ...staff.*
Actions: *

Add new Permission

Define class, name and actions of the new Permission

Specify the class: JDOMakePersistentPermission

Specify the name: company.staff.*

Specify the actions:

Figure 10: Assignment of Permissions and Roles using the RBAC Management Mode

Version 1.0.1, http://www.jcp.org/en/jsr/detail?id=12, 2004.

[7] Java Community Process. JSR-243: Java Data Objects 2.0 - An Extension to the JDO specification, Final Release, Version 2.0, http://www.jcp.org/en/jsr/detail?id=243, 2006.

[8] A. Korthaus and M. Merz. A Critical Analysis of JDO in the Context of J2EE. In A.-A. Ban, H. Arabnia, and M. Youngsong, editors, *International Conference on Software Engineering Research and Practice (SERP '03)*, volume I, pages 34–40. CSREA, 2003.

[9] M. Merz. JDOSecure: A Security Architecture for the Java Data Objects-Specification. 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), 6.-8. July, Los Angeles, California, USA, 2006.

[10] M. Merz. Using the Dynamic Proxy Approach to Introduce Role-Based Security to Java Data Objects. 18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06), 5.-7. July, San Francisco, California, USA, 2006.

[11] S. Oaks. *Java Security*. The Java Series. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 2001.

[12] Sun Microsystems. JAAS Login Configuration File. http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/LoginConfigFile.html, 2002.

[13] Sun Microsystems. *The Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[14] TheServerSide.COM. Craig Russell Responds to Roger Sessions' Critique of JDO. http://www.theserverside.com/articles/article.tss?l=RusselvsSessions, 2001.

[15] TheServerSide.COM. A Criticism of Java Data Objects (JDO). http://www.theserverside.com/news/thread.tss?thread_id=8571, 2003.

An Extensible Mechanism for Long-Term Persistence of JavaBeans Components

Chien-Min Wang[1], Shun-Te Wang[1], Hsi-Min Chen[2] and Chi-Chang Huang[1]
[1] Institute of Information, Academia Sinica
Nankang 115, Taipei, Taiwan, R.O.C.
[2] Department of Computer Science and Information Engineering
National Central University, Taoyuan, Taiwan, R.O.C.
{cmwang, wangsd, seeme, hunter}@iis.sinica.edu.tw

ABSTRACT
Long Term Persistence for JavaBeans (LTP) is an API that supports a general mechanism for serializing JavaBeans into an XML-based text format and vice versa. As Java programming language does not currently support *orthogonal persistence*, a programmer can choose to convert the internal state of an application into a permanent storage and vice versa using the LTP API. In this paper, we propose a mechanism that is extensible and optional for LTP, without modifying the LTP specification, to maximize the flexibility and extendibility of Java applications. Our approach embeds *scripts* in an encoded XML text in order to reconstruct missing objects, or to create some additional helper objects during the subsequent decoding process. Moreover, as the decoding process may take too much time to instantiate certain heavyweight objects, the proposed mechanism also supports an optional cache pool to speed it up.

Categories and Subject Descriptors
I.7.2 [**XML**]: Document Preparation – *format and notation, languages and systems, scripting languages.* D.3.2 [**Java**]: Language Classifications – *extensible languages, object-oriented languages*

General Terms
Algorithms, Performance, Design, Experimentation, Languages.

Keywords
Persistence, JavaBeans, LTP, Java, XML, scripting, cache, serialization.

1. INTRODUCTION
Object persistence has become an important requirement of modern object-oriented programming languages like C# and Java [11]. JavaBeans are reusable software components written in Java, and designed to be manipulated visually by an integrated development environment (IDE) for building applications [20]. Long Term Persistence for JavaBeans (LTP) is a persistence model for JavaBeans components [16]. The LTP specification was developed through the Java Community Process, and has been included in Java 2 Standard Edition since release 1.4. The LTP persistence scheme can convert a *tree* or *graph* of JavaBeans objects into an XML-based text format and vice versa.

Currently, the Java programming language does not support *orthogonal persistence* [14][17], which would preserve the states of any objects independent of their classes. Hence, a programmer can choose to convert the internal state of an application into a permanent storage and vice versa using the LTP API.

The LTP API provides two critical classes for JavaBeans persistence: the *XMLEncoder* and the *XMLDecoder*. An instance of the XMLEncoder class is used to write text files representing JavaBeans components. The XMLEncoder clones the object graphs and records the steps taken so that the clone can be recreated at a later time. The XMLDecoder can read the XML documents created by the XMLEncoder, parse all the XML data of the given input stream, rebuild the object graphs, and place all the objects in a queue. When the *readObject()* method is called, an object is simply popped off the queue and returned.

The use of the LTP API implies that any object can be made persistence dependent on its data type when encoded. For this purpose, the current implementation of the LTP API contains a set of *PersistenceDelegates*, organized according to the class of object. If there is no PersistenceDelegate for a given class, an instance of the *DefaultPersistenceDelegate* class is used to provide a default encoding strategy. However, *exceptions* may occur because, in this case, an object is assumed to be a JavaBean that follows the idioms laid out in the JavaBeans specification. In addition, the current LTP encoder cannot handle the persistence of anonymous inner classes, which results in *object reference losses*. This may lead to problems because the object graph is incomplete.

Another interesting issue occurs when a class is implemented using the *Singleton* design pattern [10]. The current XMLEncoder/XMLDecoder paradigm cannot guarantee that a Singleton object will remain a true Singleton, i.e., not two distinct objects, during the decoding process. The reason is that a properly created Singleton cannot be saved using the XMLEncoder due to the absence of a public constructor for the Singleton. In other words, the current XMLEncoder/XMLDecoder paradigm is designed to save JavaBeans that are not meant to be Singletons.

In this paper, we propose a mechanism that is extensible and optional for LTP, without modifying the LTP specification, the Java run-time system, or the Java language itself, to maximize the flexibility and extendibility of Java applications. Our approach embeds *scripts* in the encoded XML text in order to recover missing *object references* during the subsequent decoding process. Any Singleton object can be created in this manner. Moreover, as the decoding process may take too much time to instantiate certain heavyweight objects, the proposed mechanism supports an optional cache pool to speed it up.

The remainder of this paper is organized as follows. Section 2 describes related work. The design concept of the mechanism is proposed in Section 3. Section 4 presents the API implementation of the proposed mechanism, which is followed by a case study in section 5. We then present our conclusions in Section 6.

2. RELATED WORK

Software developers using object-oriented languages sometimes need to make their objects *persistent*, e.g., software agents [5]. The definition of *persistence* is a little vague in different application and research areas. In general, *serialization*, i.e., *Object Persistence*, means that an object graph is regarded as open data that can be converted into bits of code and vice versa [11]. Serialization also means that an object's life cycle is not restricted to the program or session that created it. In the Java programming language, serialization in a binary format is used for lightweight persistence, or for marshalling arguments of Remote Method Invocation (RMI) calls [6][15][19]. On the other hand, serialization in XML format can be used in Web services.

A comparative study of both binary and XML object persistence for Java and .NET platforms can be found in [11]. From a performance perspective, binary persistence outperforms XML persistence in terms of memory space and processing time. XML persistence, however, is readable by humans and more portable than binary persistence. Because XML uses Unicode, UTF-8 or ASCII as the presentation format for data, a SOAP-based Web service may suffer from severe performance degradation when sending scientific data. A technique called differential serialization that helps alleviate the SOAP performance problem is described in [1].

In [14], the authors propose a systematic method for developing a multi-language system with orthogonal persistence, in which the necessary data conversion between languages is transparent to users. Meanwhile, the authors of [12] investigate how reachability-based orthogonal persistence can be supported without modification of the compiler.

Binary serialization in Java was initially used to support RMI, which allows arguments to be passed between two virtual machines. RMI works well when two virtual machines contain compatible class versions. In [15] and [19], the authors study approaches that improve the performance of RMI, while the authors of [6] discuss how binary serialization can be used to marshal general communication data in Java parallel programming. The authors of [4] implement their object serialization protocol with the Java Native Interface to obtain higher throughput of parallel applications using RMI. In addition, [18] proposes a serialization algorithm that reduces the size of serialized objects, which is useful in many distributed applications to prevent a significant degradation in performance.

To support transactions and general-purpose persistence, a Java virtual machine called *Jest* is proposed in [9]. In [2], the authors describe a standard, called ODMG Java binding, which considers fundamental database capabilities such as *transactions* and *queries* for robust applications that need object persistence.

Our work focuses on XML object persistence in Java, and follows the standard LTP specification without modifying the Java run-time system or the Java language. We propose and implement an extensible mechanism that allows users to write scripts and use cached objects. The mechanism is functionally compatible with the standard LTP, and enhances the flexibility and extendibility of Java applications.

3. DESIGN CONCEPT

This section describes the design concept of the extensible mechanism, which comprises three parts. The first is the provision of a script capability to enhance the original XMLEncoder of LTP API, so that users should be able to define additional variables and add scripts to the XML text when necessary during the encoding process. The second part is the provision of a script-running environment, and the third part provides a cacheable object pool for the decoding process.

3.1 Provision of a script capability for the XMLEncoder

The current XMLEncoder cannot handle the persistence of anonymous classes or certain inner classes, which results in *the loss of object references*. For example, if an instance of the *javax.swing.JFrame* class has a *WindowListener* object implemented using an anonymous class, then, when being encoded, a *java.lang.IllegalAccessException* will be thrown because the XMLEncoder cannot access anonymous members of a given class. In this scenario, the XMLEncoder just discards the object and continues to encode the other object instances in the object graph.

To deal with this problem, users can recover a missing event listener for a *javax.swing.JFrame* object referred by a variable ***frame1***, for example, during the decoding process by running a script like the following:

```
//<SCRIPT LANGUAGE="Java"
//<!--------------------------------
   import java.awt.*;
   import java.awt.event.*;
   import javax.swing.*;

   frame1.addWindowListener(new WindowAdapter(){
     public void windowIconified(WindowEvent e){
       // do something
     }
   });
//--------------------------------->
//</SCRIPT>
```

The script is encoded and embedded in the *runScript* method, as shown in Figure 1. Note that, in this script, the *WindowAdapter* class implements the *WindowListener* interface. Furthermore, in the encoding process, this script must be embedded in the XML text immediately after the instance of the *javax.swing.JFrame* class is written by the *writeObject()* method of the XMLEncoder.

To embed *scripts* in the encoded XML text for the recovery of missing *object references* in the decoding process, a Java source interpreter can be applied. In fact, the interpreters of any scripting language that can interact with Java virtual machine are suitable for this task. The *BeanShell* Scripting Language [3], *DynamicJava* pure Java source interpreter [8] and *Jython* [13] are examples.

Note that if a script-embedded XML text is used for argument passing of Web services or remote procedure calls, classes defined in the script do not need to be compiled and deployed in the *classpath* at both the client side and the server side. This makes applications more dynamic and flexible at runtime.

Every object written by the XMLEncoder uses a standard XML schema. According to the schema, when an object graph contains cycles or multiple references to the same object, a name, i.e., an *identifier* is given to the object so that it can be referred to later. An identifier is created using the *id* attribute, which binds a name to an object instance. A reference is made to a named instance by using an *idref* attribute in an element with the <object> tag. In other words, the XMLEncoder has its own name space.

However, users cannot know which identifier will be assigned to a given object instance before the object instance is written by the *writeObject()* method of the XMLEncoder. Hence, users cannot write scripts to access the object instance in advance. Moreover, a Java source interpreter has its own name space to maintain its object pool. So, our mechanism provides a name mapping table to link the two name spaces.

For example, when an object instance of the *javax.swing.JFrame* class is written, our mechanism first looks for the name of the instance in the name spaces of the XMLEncoder and the Java interpreter and appends a ***Hashtable*** to map the two name spaces. It then writes the script for the XML text, as shown in Figure 1. In this way, users may write the following Java codes to persist a *javax.swing.JFrame* instance referred by the variable f0:

```
 … f0 = new JFrame(); …
   encoder.getInterpreter().defineVariable
                                   ("frame1", f0);
   encoder.writeObject(f0, script); ….
```

Thus, users do not need to know the assigned *identifier* of an object instance before they write scripts.

3.2 Provision of a script-running capability for the XMLDecoder

The standard XMLDecoder class is used in the same way as the *ObjectInputStream*. When being instantiated, the XMLDecoder simply reads the XML documents created by the XMLEncoder, and parses the XML text to reconstruct the whole object tree. The API document of the XMLDecoder describes that the *readObject()* method can be used to read the next object from the underlying input stream; however the description is not very precise. After carefully reviewing the source code of the XMLDecoder class provided in J2SE 1.4 or 1.5, we found that the *readObject()* method just dequeues an object for a method call. Therefore, after the constructor of the XMLDecoder class has been called and returned, all object instances of the whole object tree have already been created and initialized. That is, if users write the Java code

```
      Object f = decoder.readObject();
```

to read an object, this method just returns a created object instance from the front of the queue.

Note that, in the decoding process, a *PersistenceDelegate* cannot be used because it is designed to control all aspects of an object's persistence for the XMLEncoder in the encoding process. In addition, since the XMLDecoder parses the whole XML text immediately after its constructor has been called, all additional embedded methods, such as the *linkLocalPool* and *runScript* methods depicted in Figure 1, will be executed during the constructor's running time. This, however, may cause a problem in that, when a user wants to read an object instance from the XMLDecoder, the decoding process actually runs the corresponding script for the object beforehand.

To deal with this problem, our mechanism prohibits script-running during decoding and records the object-script map in a hash table for the Java source interpreter to enable script-running. Therefore, the corresponding script is only executed when an object instance is read by the user. In addition, the proposed mechanism also establishes and maintains an object pool when decoding. As shown in Figure 1, this is done when the *linkLocalPool* method is executed.

```
…
<object id="JFrame0" class="javax.swing.JFrame"/>
…
<void method="linkLocalPool">
  <object class="java.util.Hashtable">
    <void method="put">
     <string>frame1</string>                   Mapping of
     <object idref="JFrame0"/>                 Name Spaces
    </void>
  </object>
</void>

<void method="runScript">
 <string>
//&lt;SCRIPT LANGUAGE=&quot;Java&quot;&gt;
//&lt;!--------------------------------
  import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;

  frame1.addWindowListener(new WindowAdapter(){
    public void windowIconified(WindowEvent e){
      // do something
    }   });
//----------------------------------&gt;
//&lt;/SCRIPT&gt;
 </string>
</void> ….
```

Figure 1. Linking object pool.

3.3 Provision of a cache pool for the XMLDecoder

The decoding process may take too long to instantiate certain heavyweight objects, but providing a caching system for Java applications is usually a feasible way to speed up the process. In addition, for some special applications, a certain class may be implemented as a Singleton design pattern. Nevertheless, the current XMLEncoder/XMLDecoder paradigm cannot ensure that an object will remain a true Singleton, i.e., two or more distinct objects will not be allowed during the decoding process. To deal with these problems, our mechanism supports an optional cache pool to allow the XMLDecoder not to instantiate a new object, but to use an existing object instance instead. For example, users of our mechanism can write the following Java codes to persist a UserClass object:

```
UserClass s = new UserClass();
encoder.setPersistenceDelegate(UserClass.class,
                     new UserClassDelegate() );
encoder.getInterpreter().defineVariable
                                ("myUserClass", s);
encoder.setCacheEnabled(true);
encoder.writeObject(s, "// execute script here");....
```

The result is shown in Figure 2. Note that the decision about using a cached object is made in the encoding process. If the *setCacheEnabled*(***true***) method is called by the encoder, the decoder must call the corresponding *readObject*(***true***) method to use the cached object in the decoding process.

```
...
 <object class="java.util.Hashtable">
  <void method="put">
   <string>UserClass0</string>              Name of the
   <string>myUserClass </string>            Cached Object
  </void>
 </object>

 <object id="UserClass0" class="myApp.UserClass">
   <void property= ... >
  ...                                      Properties of the
   </void>                                 Cached Object
 </object>
...
  <void method="linkLocalPool">
   <object class="java.util.Hashtable">
    <void method="put">
     <string>myUserClass</string>           Mapping of
     <object idref="UserClass0"/>           Name Spaces
    </void>
   </object>
  </void>

  <void method="runScript">
   <string>// put script here</string>
  </void> ....
```

Figure 2. Enabling cache pool.

Before the encoder writes all the properties of the UserClass in the XML text file, a hash table must be written. In the decoding stage, the XMLDecoder will use this table to decide which object should not be instantiated. It will then use an existing object instance from the interpreter's object pool instead, and reset all the properties of the UserClass object. For users to get the UserClass object with new properties in the decoding process, the Java codes may look like this:

```
UserClass s = new UserClass();
decoder.getInterpreter().defineVariable
                               ("myUserClass", s);
 ...
UserClass s1 = (UserClass)
                     decoder.readObject(true); ....
```

Here, the *readObject(**true**)* method is used to notify the XMLDecoder that the object is cached and therefore it should not try to instantiate a new object. In this manner, new properties recorded in the XML text for this cached object will be set up, and then returned. In the above example, the variables *s* and *s1* refer to the same object.

Note that because a Singleton object does not have a public constructor, its instance cannot be persisted using the XMLEncoder's *writeObject()* method. On the other hand, users cannot write a *PersistenceDelegate* class for a Singleton because in order to let a *PersistenceDelegate* work, more than one instance of the class must be created, which violate the nature of a Singleton. Our mechanism can be used to solve this problem.

Users can write a wrapper class to access the Singleton object. The wrapper object is used to hold all the *properties* of the Singleton object in the encoding process so that the wrapper object is persisted. In the decoding process, users can instantiate a wrapper object bound to the Singleton, and cache it in the object pool. Hence, all *properties* stored in the wrapper object can be retrieved from the XML text and set for the Singleton object via running the script (if any) along with the wrapper object. This process is suitable for argument passing of Web services or remote procedure calls. It can also be applied when an undo/redo function is added to Java applications or Java integrated development environments.

4. API IMPLEMENTATION

This section presents the API implementation of the extensible mechanism. The key features of the implementation are (1) an interpreter is integrated with the XMLEncoder and the XMLDecoder; (2) the name spaces of the XMLEncoder and the interpreter are mapped; and (3) cached objects can be used, instead of instantiating new objects.

The result of the implemented mechanism is a Java API package, which is available for download publicly from the web site *http://sml-109.iis.sinica.edu.tw/eLTP/*.

4.1 Integration of a Java source interpreter

In order to run *scripts* in the decoding process, a Java source code interpreter must be utilized. Any interpreters that can interact with a Java virtual machine are suitable; therefore we have designed an interface *XMLScriptInterpreter* for these interpreters, as shown in Figure 3. A *concrete class* that implements this interface is regarded as a proxy of the practical interpreter.

The *XMLScriptInterpreter* interface defines five methods. The *interpret()* method is used to run the script, the two *defineVariable()* methods are used to define variables in the interpreter environment, the *setVariable()* method is used to set the value of a variable, and the *getVariable()* method is used to get the value of a specified variable.

```
interface XMLScriptInterpreter
{
  void interpret(String script);
  void defineVariable(String var, Object obj);
  void defineVariable(String var, Object obj, Class c);
  void setVariable(String name, Object value);
  Object getVariable(String name);
}
```

Figure 3. The XMLScriptInterpreter interface.

4.2 Mapping the name spaces of the encoder and interpreter

Integrating an interpreter into the current XMLEncoder implementation is necessary for the persistence of an object instance. However, trying to extend the XMLEncoder class provided by J2SE is impractical, because some inner classes of the XMLEncoder can not be accessed by its subclasses or by any classes outside the core *java.beans* package.

To solve this issue, it is not necessary to extend the XMLEncoder. Instead, it seems feasible to create a separate tool that reads the XML text produced by the XMLEncoder, and attach the script code from that. In other words, the first tool can write the XML text to the standard output using the XMLEncoder, and the second tool can read the XML text from the standard input. Users may even combine the tools on a single command line. Nevertheless, these tools must work in their respective Java virtual machines, which is not very convenient for programmers of Java applications.

Referring to the Java source code for the whole XMLEncoder package, we implement another new encoder class, called the *ExtensibleXMLEncoder*, which is functionally compatible with the XMLEncoder. Therefore, all public methods in the latter also work in the former. In addition, the following methods are added:

```
XMLScriptInterpreter getInterpreter();
void setInterpreter(XMLScriptInterpreter i);
void setCacheEnabled(boolean b);
void writeObject(Object o, String script);
void writeObject(Object o, String script,
                 boolean ifUseCache);
```

The *getInterpreter()* and *setInterpreter()* methods are used to get and assign an interpreter implementation, respectively. The *setCacheEnabled()* method indicates whether or not an existing object instance cached in the object pool can be used in the decoding process. The two *writeObject()* methods are used to persist an object instance along with a script. Note that all the above methods are optional. If none of them are used, the *ExtensibleXMLEncoder* will behave exactly the same as the XMLEncoder.

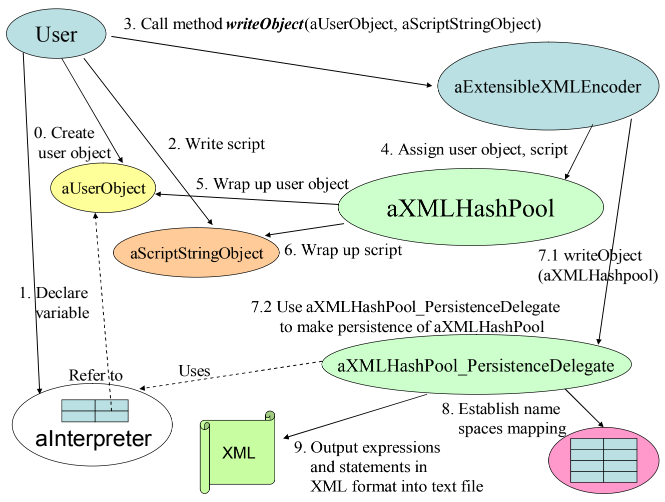
Figure 4 shows the procedure for conducting the persistence of a given object with its corresponding attached script. A user of the *ExtensibleXMLEncoder* first creates a *UserObject* that will be persisted, and declares a variable name for the object in the interpreter using the interpreter's *defineVariable()* method. The user then writes a *ScriptStringObject* to accompany the object. Next, the *writeObject()* method is used to persist the object and the script. From a programmer's perspective, the encoding process is finished after this method has been called.

To map the name spaces of the encoder and interpreter, a wrapper class called the *XMLHashPool* is designed as a JavaBean to hold the *references* of the object, the script and the interpreter. In fact, after the user writes the object and the script, the *ExtensibleXMLEncoder* wraps them using an *XMLHashPool* object and then calls the writeObject() method of the standard XMLEncoder to persist the *XMLHashPool* object. Because the *XMLHashPool* object is a JavaBean, the XMLEncoder will archive graphs of the JavaBean as textual representations of the JavaBean's public properties. Note that the XMLEncoder uses a delegation model to have its behavior controlled independently of itself. Any class that extends the PersistenceDelegate abstract class takes responsibility for expressing the state of an instance of a given class in terms of the methods in the class's public API. For the *XMLHashPool*, hence, we implement an *XMLHashPool_PersistenceDelegate* class, which can establish name space mapping, and output *expressions* and *statements*, e.g., the body of the *linkLocalPool* method, in XML format, as depicted in Figure 1.

Note that technically it is possible to obtain the name of an instance existing in the XMLEncoder, although the XMLEncoder does not provide a public method for users to do so. In our mechanism, we use the following codes to obtain the name of a given instance:

```
s = new XMLCachedExpression(obj, obj, null,
                           null).toString();
i = s.indexOf("=");
if (i != -1) s = s.substring(0, i);
```

Here, the *XMLCachedExpression* class that works with the *ExtensibleXMLEncoder* is a clone of the standard Expression class in J2SE. The *toString()* method returns a string in the form of *name=name.method()*. We just take the element on the left of the equal sign. The resulting substring *s* is the name of the *obj* object in the XMLEncoder. This is the *identifier*, i.e., the *id* attribute or *idref* attribute noted in the XML text.



User
3. Call method writeObject(aUserObject, aScriptStringObject)
aExtensibleXMLEncoder
0. Create user object
2. Write script
4. Assign user object, script
aUserObject
5. Wrap up user object
aXMLHashPool
aScriptStringObject
6. Wrap up script
7.1 writeObject (aXMLHashpool)
1. Declare variable
7.2 Use aXMLHashPool_PersistenceDelegate to make persistence of aXMLHashPool
Refer to
Uses
aXMLHashPool_PersistenceDelegate
aInterpreter
XML
8. Establish name spaces mapping
9. Output expressions and statements in XML format into text file

Figure 4. Procedure for persistence and name space mapping.

4.3 Object caching, decoding, and script running

Based on the Java source code for the whole XMLDecoder package in J2SE, we implement another new decoder class called the *ExtensibleXMLDecoder*, which is functionally compatible with the XMLDecoder. In other words, all public methods found in the latter also work in the former. Moreover, the following methods are added:

```
XMLScriptInterpreter getInterpreter();
void setInterpreter(XMLScriptInterpreter i);
Object readObject(boolean ifUseCache);
```

These methods are also optional. If none of them are applied, the *ExtensibleXMLDecoder* will operate in exactly the same way as the XMLDecoder. Here, users can get and assign an interpreter implementation using the *getInterpreter()* and *setInterpreter()* methods, respectively. The *readObject()* method is used to read back a persisted object.

As shown in Figure 2, after decoding the XML text, the *ExtensibleXMLDecoder* knows from the recorded hash table that the object, identified by the *id* attribute, does not need to be instantiated, provided that the argument value passed to the *readObject()* method is *true*. If this is so, the *properties* of the object recovered from the XML text will be set for the existing object cached in the interpreter's object pool, instead of a new object instance. Once the *ExtensibleXMLDecoder* determines the instance to be used, it runs the corresponding script using the interpreter.

Note that name mapping for the interpreter is done by the *linkLocalPool* method during decoding. In addition, assignment of the corresponding script for an object is done by the *runScript* method, as shown in Figure 2.

The default design of the XMLDecoder instantiates a new object instance every time the object's *expression*, represented by the *element* with the <*object*> tag, is processed. However, the current standard XMLDecoder does not provide any public methods that enable users to prevent the creation of new objects. Technically, it is feasible to let the decoder use a cached object. To do so, in our mechanism, we implement a class called the *XMLCachedMutableExpression* that extends a clone of the MutableExpression class in J2SE to work for the *ExtensibleXMLDecoder*.

After carefully reviewing the source code of the MutableExpression class, we found that the method named *invoke()* is used by the decoder to instantiate an object instance of a given class. Therefore, the *invoke()* method of the MutableExpression class is enhanced to provide the option of returning a cached object instance. Its algorithm is as follows:

```
Object invoke() {
  if ( "new".equals( getMethodName() ) ){
     String s0 = getBoundName();
     String s1 =
          find_Name_Mapping_in_Interpreter( s0 );
     return interpreter.getVariable( s1 );
  }
  return super.invoke();
}
```

If the *getMethodName()* method returns "*new*", then an object instance has to be returned. However, in this case we try to find the *identifier* of the given object using the *getBoundName()* method. According to the *identifier*, the given object's corresponding cached object instance can be found in the object pool of the interpreter, and then returned. If no cached object exists in the pool, the *invoke()* method returns *null*, and a new object is instantiated.

5. CASE STUDY

In this section, we present an example to show how JavaBeans can be persisted using our extensible mechanism. Without loss of generality, we choose the *LogoAnimator2* and *ColorSliderPanel* JavaBeans from the book *Advanced Java 2 Platform: How to Program* [7].

Because users of an application may make mistakes, as programmers it is important that we help them recover gracefully by including support for the Undo/Redo functionality in the user interfaces of the application. The following case study demonstrates how to add Undo/Redo capabilities to an existing Java application. As shown in Figure 5, a user can determine the RGB values for various colors using the *ColorSliderPanel* JavaBean, and see the resulting color immediately in the *LogoAnimator2* JavaBean. The user can *save* the current state of the application, i.e., the current color, as depicted in Figure 5(a), and then, choose or test another color if necessary, as shown in Figure 5(b). The application can go back to the previous saved state if the user *loads* the persisted state, as shown in Figure 5(c).

Note that, when performing the *load* operation, the persisted state stored in an XML text file will be decoded. However, current object instances of the *LogoAnimator2* and *ColorSliderPanel* JavaBeans are still working, so there is no need to instantiate new object instances. The *LogoAnimator2* and *ColorSliderPanel* JavaBeans can be cached to let the *ExtensibleXMLDecoder* recover their states and properties. The script embedded in the XML text is used to synchronize the *LogoAnimator2* and *ColorSliderPanel* JavaBeans. A snapshot of the XML file is shown in Figure 6, and the main program is given in Figure 7.

In addition to this example, the Java software package that implements the proposed mechanism, as well as other examples of its usage can be downloaded free-of-charge from http://sml-109.iis.sinica.edu.tw/eLTP/.

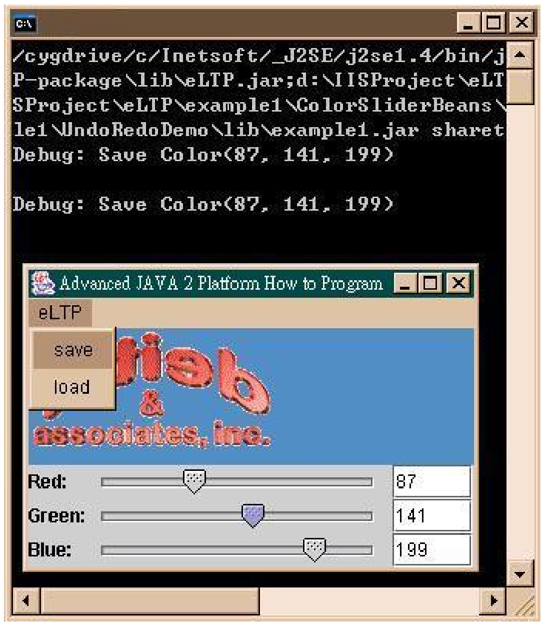
6. CONCLUSIONS

In this paper, we have focused on serializing Java objects into the XML format. In order to obtain more flexibility and extendibility of Java applications using the LTP API, we propose a mechanism that follows the standard LTP specification without modifying the Java run-time system or the Java language. Our approach allows *scripts* to be embedded in an XML text for the reconstruction of certain missing objects, or for the creation of some additional helper objects in the decoding process. In addition, as the decoding process may take too much time to instantiate certain heavyweight objects, the proposed mechanism also supports an optional cache pool to speed up the process.

The proposed mechanism is compatible with the interpreters of any current scripting language that can interact with Java virtual machine. Moreover, the mechanism can make applications more dynamic and flexible at runtime. That is, if the script-embedded XML text is used for argument passing of Web services or remote procedure calls, classes defined in the script do not need to be compiled and deployed in the *classpath* at both client side and server side.

7. ACKNOWLEDGMENTS

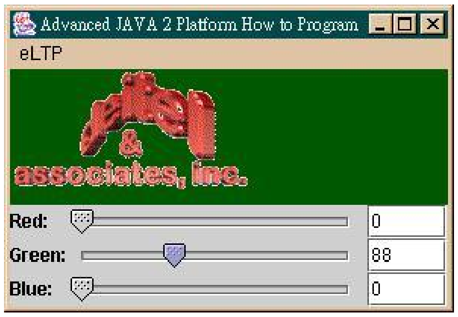
This work was supported by National Science Council under Contract No. NSC94-2213-E-001-023.



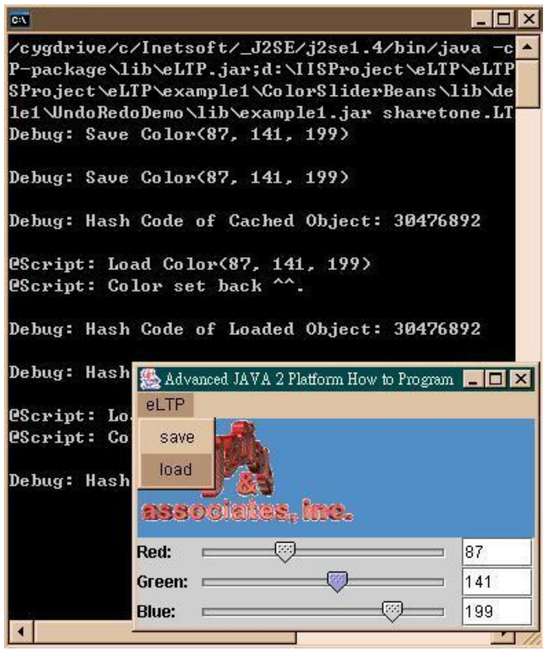
```
/cygdrive/c/Inetsoft/_J2SE/j2se1.4/bin/j
P-package\lib\eLTP.jar;d:\IISProject\eLT
SProject\eLTP\example1\ColorSliderBeans\
le1\UndoRedoDemo\lib\example1.jar sharet
Debug: Save Color(87, 141, 199)

Debug: Save Color(87, 141, 199)
```

(a)



(b)



```
/cygdrive/c/Inetsoft/_J2SE/j2se1.4/bin/java -c
P-package\lib\eLTP.jar;d:\IISProject\eLTP\eLTP
SProject\eLTP\example1\ColorSliderBeans\lib\de
le1\UndoRedoDemo\lib\example1.jar sharetone.LT
Debug: Save Color(87, 141, 199)

Debug: Save Color(87, 141, 199)

Debug: Hash Code of Cached Object: 30476892

@Script: Load Color(87, 141, 199)
@Script: Color set back ^^.

Debug: Hash Code of Loaded Object: 30476892

Debug: Hash

@Script: Lo
@Script: Co

Debug: Hash
```

(c)

Figure 5. The Undo/Redo model.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_02" class="java.beans.XMLDecoder">
 <object class="java.util.Hashtable">
  <void method="put">
   <string>ColorSliderPanel0</string>
   <string>colorpane</string>
  </void>
 </object>
 <object id="ColorSliderPanel0"
class="com.deitel.advjhtp1.beans.ColorSliderPanel">
  <void property="redGreenBlue">
   <void index="0">
    <int>87</int>
   </void>
   <void index="1">
    <int>141</int>
   </void>
   <void index="2">
    <int>199</int>
   </void>
  </void>
 </object>
 <object class="sharetone.LTP.ScriptLTP.XMLHashPool">
  <void method="linkLocalPool">
   <object class="java.util.Hashtable">
    <void method="put">
     <string>colorpane</string>
     <object idref="ColorSliderPanel0"/>
    </void>
   </object>
  </void>
  <void method="runScript">
   <string>
//&lt;SCRIPT LANGUAGE=&quot;DynamicJava&quot;&gt;
//&lt;!--------------------------------
import com.deitel.advjhtp1.beans.SliderFieldPanel;
import javax.swing.JPanel;

System.out.print(&quot;@Script:   Load   Color(&quot;   +
colorpane.getRedGreenBlue(0) + &quot;, &quot;);
System.out.print(colorpane.getRedGreenBlue(1)  +  &quot;,
&quot;);
System.out.println(colorpane.getRedGreenBlue(2)         +
&quot;)&quot;);
Object[] o = colorpane.getComponents();
Object[] o2 = ((JPanel) o[1]).getComponents();

for(int i = 0; i &lt; o2.length; i++)
   ((SliderFieldPanel)
o2[i]).setCurrentValue(colorpane.getRedGreenBlue(i));

System.out.println(&quot;@Script: Color set back ^^.   \n
&quot;);
//---------------------------------&gt;
//&lt;/SCRIPT&gt;
</string>
  </void>
 </object>
</java>
```

Figure 6. A snapshot of the XML file.

```
…
import sharetone.LTP.ExtensibleXMLDecoder;
import sharetone.LTP.ExtensibleXMLEncoder;
import com.deitel.advjhtp1.beans.ColorSliderPanel;
import com.deitel.advjhtp1.beans.LogoAnimator2;
…
public class GuiMain implements ExceptionListener, ColorListener, ActionListener {
   public static void main(String[] args) throws Exception{ … }

   public void exceptionThrown(Exception e){}                      // implement ExceptionListener
   public void colorChanged(ColorEvent colorEvent){ … }            // implement ColorListener
   public void actionPerformed(ActionEvent evt){ … }               // implement ActionListener

   private ColorSliderPanel cpane = null;
   private LogoAnimator2 logo = null;
   private ExceptionListener lis = null;
   private JFrame frame = null;

   public GuiMain(ColorSliderPanel cpane, LogoAnimator2 logo, JFrame frame) { … }

    private void save(ColorSliderPanel cpane, LogoAnimator2 logo, ExceptionListener lis) throws Exception {
         OutputStream os = new BufferedOutputStream(new FileOutputStream("XMLtext.xml"));
         ExtensibleXMLEncoder encoder = new ExtensibleXMLEncoder(os);
         encoder.setExceptionListener(lis);
         encoder.getInterpreter().defineVariable("colorpane",  cpane);
         encoder.setCacheEnabled(true);  // use cached object in future decoding process
         System.out.println("Debug: Save Color(" + cpane.getRedGreenBlue(0)+ ", "
                                       + cpane.getRedGreenBlue(1)+ ", " + cpane.getRedGreenBlue(2) + ")\n");

         String script = "\n//<SCRIPT LANGUAGE=\"DynamicJava\">                                    \n";
         script = script + "//<!---------------------------------                                \n";
         script = script + "import com.deitel.advjhtp1.beans.SliderFieldPanel;                  \n";
         script = script + "import javax.swing.JPanel;                                          \n";
         script = script + "                                                                    \n";
         script = script + "System.out.print(\"@Script: Load Color(\" + colorpane.getRedGreenBlue(0) + \", \");  \n";
         script = script + "System.out.print(colorpane.getRedGreenBlue(1) + \", \");             \n";
         script = script + "System.out.println(colorpane.getRedGreenBlue(2) + \")\");            \n";
         script = script + "                                                                    \n";
         script = script + "Object[] o = colorpane.getComponents();                             \n";
         script = script + "Object[] o2 = ((JPanel) o[1]).getComponents();                      \n";
         script = script + "                                                                    \n";
         script = script + "for(int i = 0; i < o2.length; i++)                                  \n";
         script = script + "   ((SliderFieldPanel) o2[i]).setCurrentValue(colorpane.getRedGreenBlue(i));    \n";
         script = script + "                                                                    \n";
         script = script + "System.out.println(\"@Script: Color set back ^^.   \\n    \");          \n";
         script = script + "//---------------------------------->                               \n";
         script = script + "//</SCRIPT>                                                         \n";
         encoder.writeObject(cpane, script);
         encoder.flush();
         encoder.close();
      }

      private void load() throws Exception {
         InputStream is = new BufferedInputStream(new FileInputStream(" XMLtext.xml"));
         ExtensibleXMLDecoder decoder = new ExtensibleXMLDecoder(is);
         System.out.println("Debug: Hash Code of Cached Object: " + this.cpane.hashCode() + "\n");

         decoder.getInterpreter().defineVariable("colorpane",  this.cpane);     // to cache this object
         ColorSliderPanel cpane1 = (ColorSliderPanel) decoder.readObject(true); // use cached object instance
         System.out.println("Debug: Hash Code of Loaded Object: " + cpane1.hashCode() + "\n");
      }
}
```

Figure 7. The main program.

8. REFERENCES

[1] Abu-Ghazaleh, N., Lewis, M. J., and Govindaraju, M., Differential serialization for optimized SOAP performance, In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)* (June 4-6, 2004), 55-64.

[2] Barry, D., and Stanienda, T., Solving the Java object storage problem, *IEEE Computer Magazine*, 31, 11 (November 1998), 33-40.

[3] BeanShell - Lightweight Scripting for Java. http://www.beanshell.org/

[4] Breg, F., and Polychronopoulos, C. D., Java virtual machine support for object serialization, In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande* (Palo Alto, California, USA, 2001), 173-180.

[5] Buhler, P. A., and Huhns, M. N., Trust and persistence [software agents], *IEEE Internet Computing Magazine*, 5, 2 (March-April, 2001), pp. 85-87.

[6] Carpenter, B., Fox, G., Ko, S. H., and Lim, S., Object serialization for marshalling data in a Java interface to MPI, In *Proceedings of the ACM 1999 conference on Java Grande* (San Francisco, California, USA, 1999), 66-71.

[7] Deitel, H. M., Deitel, P. J., and Santry, S. E., *Advanced Java 2 Platform: How to Program*, Prentice-Hall Inc., 2002, 340-379.

[8] DynamicJava - Java source interpreter. http://koala.ilog.fr/djava/

[9] Garthwaite, A., and Nettles, S., Transactions for Java, In *Proceedings of the International Conference on Computer Languages (ICCL'98)* (Chicago, IL, USA, May 14-16, 1998), 16-27.

[10] Grand, M., *Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*, 2nd Edition, John Wiley & Sons, 2002, ISBN: 0471227293.

[11] Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S., and Zivkovic, A., Object serialization analysis and comparison in Java and .NET, *ACM SIGPLAN Notices*, 38, 8 (Aug. 2003), 44-54.

[12] Hosking, A. L., and Chen, J., Mostly-copying reachability-based orthogonal persistence, In *Proceedings of the 14th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'99)* (Denver, Colorado, USA, November 1-5, 1999), 382-398.

[13] Jython - Python language written in Java. http://www.jython.org/

[14] Kato, K., and Ohori, A., An approach to multilanguage persistent type system, In *Proceedings of the 25h Hawaii International Conference on System Sciences*, Vol. II (Kauai, Hawaii, USA, January 7-10, 1992), 810-819.

[15] Kono, K., and Masuda, T., Efficient RMI: dynamic specialization of object serialization, In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)* (Taipei, Taiwan, April 10-13, 2000), 308-315.

[16] Long Term Persistence of JavaBeans Components: XML Schema. http://java.sun.com/products/jfc/tsc/articles/persistence3/

[17] Nettles S., and O'Toole, J., Implementing orthogonal persistence: a simple optimization using replicating collection, In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS'93)* (Asheville, NC, USA , December 9-10, 1993), 177-181.

[18] Opyrchal, L., and Prakash, A., Efficient object serialization in Java, In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/ Middleware* (Austin, TX, USA, May 31-June 4, 1999), 96-101.

[19] Park, J. G., and Lee, A. H., Specializing the Java object serialization using partial evaluation for a faster RMI [remote method invocation], In *Proceedings of the Eighth International Conference on Parallel and Distributed Systems (ICPADS 2001)* (June 26-29, 2001), 451-458.

[20] The Bean Builder. https://bean-builder.dev.java.net/

Heap protection for Java virtual machines

Yuji Chiba
Systems Development Laboratory, Hitachi, Ltd.
1099 Ozenji, Asao, Kawasaki,
Kanagawa, Japan
yuji@sdl.hitachi.co.jp

ABSTRACT

Java virtual machine (JVM) crashes are often due to an invalid memory reference to the JVM heap. Before the bug that caused the invalid reference can be fixed, its location must be identified. It can be in either the JVM implementation or the native library written in C language invoked from Java applications. To help system engineers identify the location, we implemented a feature using page protection that prevents threads executing native methods from referring to the JVM heap. This feature protects the JVM heap during native method execution, and when native method execution refers to the JVM heap invalidly, it interrupts the execution by generating a page-fault exception and then reports the location where the page-fault exception was generated. This helps the system engineer to identify the location of the bug in the native library. The runtime overhead for using this feature averaged 4.4% based on an estimation using the SPECjvm98, SPECjbb2000, and JFCMark benchmark suites.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*memory management*

General Terms

Reliability, Languages

1. INTRODUCTION

Servers are being equipped with more and more memory, and some server applications, such as application servers, now use several gigabytes of memory for the heap. Therefore, if a server application is executed on a 32-bit operating system (OS) that provides a 4-Gigabyte (GByte) logical address space for its processes, the server application commits tens of percent of its logical address space as its heap. This reduces the use of page protection, which has long been used to detect invalid memory references, because most of the logical address space is committed as heap and is thus valid for reference.

Developers of Java™[1] runtime environments (JREs) often face this problem because server applications running on them sometimes crash due to corrupted heap. If the engineer trying to debug the problem does find that the heap is corrupted, he or she will likely ask the JRE developer to perform the debugging, simply because the heap has been allocated by the JRE. The engineer will not suspect that the bug is in a server application if the application is implemented in Java, because Java does not permit invalid memory references. While the bug may be in native libraries used by the server applications, the engineer may not ask a native library developer to perform the debugging because it is often not easy for an engineer to tell which native library contains the bug. This is because JRE may crash long after the bug corrupted the heap, so all that the engineer can tell from the core file of the crashed JRE process is that the heap is corrupted and that it had been allocated by the JRE.

To cope with this problem, we implemented a JRE that uses page protection to protect its heap from invalid references due to bugs in the native libraries. The JRE protects its heap during native method execution, and when the native method execution refers to the heap invalidly[2], it interrupts the execution by generating a page-fault exception and then reports the location where the page-fault exception was generated. This helps the engineer to identify which native library contains the bug.

This heap protection feature permits a thread to write to the heap only when the thread is executing a Java method. When the thread calls a native method through the Java Native Interface[21] (JNI), it is not granted permission to write to the heap because our JNI implementation deprives the thread of this permission. This protection feature controls permission to write to the heap by thread, while mainstream operating systems such as Windows®[3] and Linux®[4] control

[1]Java and HotSpot are trademarks of Sun Microsystems, Inc. in the United States and other countries.

[2]Basically, native methods should refer to the JRE heap via the Java Native Interface (JNI). Any direct reference is invalid if the JNI implementation does not permit native methods to refer directly to the body of a Java array, as that of HotSpot VM (`jdk1.5.0_06`) does not.

[3]Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

[4]Linux is a trademark or a registered trademark of Linus Torvalds in the United States and other countries.

permission by process. In other words, they provide only one protection domain[5] to each process, not to each thread. Thus the protection domain supported by these operating systems is not appropriate for implementing our heap protection feature. We thus modified Linux and implemented a per-thread protection domain for heap. We call it the 'heap protection domain'. It is used as follows.

1. If a program component (a set of procedures) uses its local heap, that is accessed only by the program component if there is no bug, the program component asks the OS to create a heap protection domain for the program component. This is usually done during the program component initialization, before the allocation of the local heap.

2. The program component then allocates the local heap, and asks the OS to give its heap protection domain permission to reference the heap.

    - A heap protection domain is a property of a process. Every thread in a process belongs to one of the process' heap protection domains. The OS permits a thread to reference memory based on the heap protection domain to which it belongs.

3. At every entry to the program component, the developer of the program component adds code asking the OS to modify the heap protection domain, so that a thread that executes the code acquires permission to refer to the heap for the program component. The code asks the OS to let the thread belong to the protection domain for the program component. The entries to the program component reside in the prologs to the public procedures of the program component and after the external procedure calls.

4. At every exit from the program component, the developer of the program component adds code asking the OS to modify the heap protection domain, so that the thread that executed the code gives up the permission to refer to the heap for the program component. The exits from the program component reside in the epilogs of the public procedures of the program component and before the external procedure calls.

Our JVM product supports a wide range of platforms (processors and operating systems), thus we want to protect the JVM heap on as many platforms as possible. Since we implemented our approach on IA32/Linux platform in April 2006, we have encountered one case of a JVM crash that we could have easily debugged using our work, but we couldn't, because the crash happened on the IA64/Linux platform. In order to prevent such crashes, operating systems have to support the heap protection domain, and this limits the practicality of our work. However, since our heap protection domain implementation requires only a conventional page protection feature provided by most processors, it can be implemented on most operating systems. We hope operating systems provide a feature like our heap protection domain in the future.

[5]A protection domain is a row of an access matrix. Every subject belongs to one of protection domains at a time, and operating systems permits a subject to access each of their resource based on the protection domain to which the subject belongs.

Using the heap protection domain, we implemented this heap protection feature on a JVM and estimated the runtime overhead. The result of this evaluation, i.e. whether the heap protection feature is practical or not, is the main contribution of this paper. To ensure that the results are reliable, we implemented the heap protection feature on a practical JVM and OS (HotSpot™virtual machine[24] (HotSpot VM) and Linux) and evaluated the runtime overhead using three benchmark suites (SPECjvm98[27], SPECjbb2000[28], and JFCMark™[6][11]) consisting of practical Java applications. To the best of our knowledge, such evaluation data has not been previously presented.

This paper is organized as follows. In section 2 we explore previous related work. In section 3 we describe our implementation of the heap protection domain, and in section 4 we describe our implementation of the heap protection feature. In section 5 we discuss the implementation of `setpd()`, which is a system call to modify the protection domain to which a thread belongs (and generates most of the runtime overhead related to using the heap protection feature), and in section 6 we evaluate the runtime overhead. In section 7 we summarize the key points.

2. RELATED WORK

Invalid memory references have plagued program developers since the dawn of computing, and many attempts have been made to eliminate them, mainly by people working in four fields: programming languages, virtual machines, hardware and operating systems.

In this section, we survey their contributions and compare them with the heap protection domain.

2.1 Programming Languages

Programming in traditional languages such as C language and assembly language enable a programmer to cast any value to a pointer and then refer to memory through the pointer, but this often results in an invalid memory reference. Modern programming languages such as Java[2] and C#[22] thus do not provide pointers. They provide references instead. References differ from pointers in that a reference may not be a raw memory address. They also provide such features as array index checking, which prevents invalid memory accesses through references. Programs written in these modern programming languages thus cannot cause invalid memory accesses.

While the modern programming languages are an effective way to avoid invalid memory accesses, not all programs that can be easily written in the modern languages, and there is vast number of existing written in traditional languages. SafeC[3] and CCured[23] insert range checks to prevent legacy C code from making invalid memory accesses, but their practicality is limited because of the runtime overhead for range checks and because their pointer implementation (a fat pointer) is not compatible with the conventional one (address only).CCured suffers less runtime overhead (30% to 150%) than SafeC (130% to 540%) because it statically verifies pointer operations and inserts a range check only if one is needed. Both SafeC and CCured implement range checks using fat pointers, thus their pointer

[6]JFCMark is a trademark of Excelsior, LLC.

implementations are not compatible with the conventional one. This means that legacy C programs that expect conventional pointer implementation do not work correctly when compiled using them. Jones and Kelly proposed a backward-compatible range check implementation, but it has even higher runtime overhead[18]. Even when pointer format compatibility is a problem, we cannot ignore the fact that recompilation itself adds costs, such as that for quality assurance.

2.2 Virtual Machines

Virtual machines[26, 9, 19] detect invalid memory accesses by monitoring the execution of memory access instructions as they interpret the machine instructions for executing the target program. While virtual machines are useful for debugging, they may not be appropriate for monitoring programs in practical use because of their large runtime overhead. Koju[19] reports that the performance of a virtual machine is 231.2 times lower than that of the machine on which it is running if the virtual machine monitors all store instructions and is implemented as an interpreter. A virtual machine implementation using dynamic translation improves performance so that it is 11.7 times lower than that of the actual machine.

Because our final goal is to monitor application server processes in practical use, approaches that significantly degrade performance are not appropriate. The heap protection domain suffers much less performance loss (up to two times lower than that of the actual machine, as described in section 6).

2.3 Hardware

Practical multiprogramming environments should provide features that prevent a program execution from invalidly accessing resources assigned to other program executions[10], and hardware and operating systems have been designed to prevent this. Most modern processors have been designed to provide address translation and page protection features.

Address translation maps logical addresses to physical addresses. This feature ensures that each program execution is given a virtually isolated address space so that it cannot access memory assigned to another program execution.

Page protection divides the address space into fixed-length spaces, or pages, and sets permissions for page access. The heap protection domain is implemented using this feature.

While most modern processors support protection by page, Mondrian Memory Protection[31] supports it by word. Such fine-grained protection should be useful for protecting fine-grained memory areas, such as words for global variables.

Hardware-level detection against bugs which misuse non-pointer values as pointers has also been investigated so far. System/38™[7] [12] assigns a tag bit to each memory word and sets the tag bit value to 1 if the value in the corresponding memory word is a pointer, so as to permit a value in a memory word to be used as a pointer to reference memory only if the tag bit value is 1.

[7]PowerPC, System/38 and i5/OS are trademarks or registered trademarks of International Business Machines Corporation in the United States and other countries.

2.4 Operating Systems

An operating system implements virtual memory and memory access control by using the address translation and the page protection features provided by hardware.

Mainstream operating systems such as Windows and Linux divide program executions into processes and assign a logical address space to each process. Because they provide memory access control to each process, not to each thread, every thread in a process has the same memory access permission. This means that the heap protection feature presented here cannot be simply implemented in these operating systems. One implementation of JVM on such an OS, MVM[8, 7], protects its heap from native method execution by dividing execution of the JVM into two processes: one executes Java applications and the other executes native methods. MVM protects a wider range of memory than our heap protection feature does, because MVM protects stacks, global variables, and heap allocated using `malloc()`[8]. However, MVM suffers large runtime overhead from JNI calls, which are implemented using inter-process communication (IPC). Czajkowski estimated that the execution time of SPECjvm98 is approximately ten times longer at most[8], if all JNI calls are implemented using IPC.

Hydra[6, 20, 32], Opal[4], and i5/OS®[15] provide memory access control features like the heap protection domain described here. i5/OS is a commercial OS currently in use, and JVMs for i5/OS[16] can easily support heap protection. However, to the best of our knowledge, there have been no reports like that presented here on the runtime overhead incurred in protecting the heap.

3. IMPLEMENTATION OF THE HEAP PROTECTION DOMAIN

We implemented the heap protection domain on Linux for IA32 processors using a page global directory (PGD), which is one of the address translation tables IA32 processors use to implement paging and page protection. This section first describes implementation of virtual memory on IA32 processors and then describes the implementation of the heap protection domain using PGD.

3.1 Virtual memory on IA32 processors

IA32 processors implement virtual memory using both segmentation and paging[13]. They first translate each logical address into a linear address using segmentation and then translate the linear address into a physical address using paging.

IA32 processors translate the linear address into a physical address as follows[9].

1. To translate linear address L, the processor first looks at its `CR3` register because it holds the PGD address.

[8]Our seven years of JVM maintenance experience has shown that many of hard-to-analyze crashes came from invalid access to the heap for Java instances or dynamically compiled codes. We thus designed our heap protection feature to protect only the heap. Our heap protection feature does not protect the other memory areas to avoid runtime overhead.

[9]IA32 processors provide two paging implementations. One uses two-level address-translation tables, and the other uses three-level address-translation tables. We describe only the former because we used it to implement the heap protection domain.

The processor then refers to the entry in the PGD for L. An entry in the PGD is called a directory entry. The processor gets the address of the directory entry for L by adding the highest ten bits of L as an offset to the PGD address.

2. The processor looks at the bit in the directory entry that indicates to what the directory entry points.

   (a) If the value of the bit is 0, the directory entry points to a page table, which is an address translation table whose entry points to a 4-KByte page. In this case, the processor translates L into the physical address as illustrated in figure 1 (a). The processor assembles the physical address from the page table entry and the lowest 12 bits of L. The processor also looks at bits in the page table entry for access control to the page.

   (b) If the value of the bit is 1, the directory entry points to a 4 MByte page (a large page). In this case, the processor translates L into the physical address, as illustrated in figure 1 (b). The processor assembles the physical address from the directory entry and the lowest 22 bits of L.

3.2 Implementation of heap protection domain using PGD

The directory entry for a large page contains bits that indicate reading or writing permission, and the processor permits reference to the page based on these bits. We implemented the heap protection domain using these bits as follows.

1. In the system call to create a heap protection domain, `createpd()`, the OS creates a PGD. For simplicity, our implementation allocates only one PGD and shares page tables among heap protection domains. Therefore, our heap protection domain can set permissions for large pages only[10]. To set permissions for 4-KByte pages, page tables should be created for each protection domain.

2. In the system call to set permissions for the large pages, `mprotect()`, the OS sets the bits in the directory entries that indicate reading or writing permission.

3. In the system call to set the protection domain for the current thread, `setpd()`, the OS looks up the pointer to the PGD for the target heap protection domain, and loads the pointer into the `CR3` register.

4. During task switching, the OS looks up the pointer to the PGD for the protection domain to which the next task belongs and loads the pointer into the `CR3` register.

4. HEAP PROTECTION IN HOTSPOT VM

In our implementation of the heap protection for HotSpot VM, we protect two heaps that HotSpot VM uses.

[10]Linux maps large pages only to address space that programs committed specially for large pages.

Heap for Java: HotSpot VM uses this heap to allocate the data structures Java applications allocate, such as Java instances.

Heap for dynamically compiled code: HotSpot VM's dynamic compiler uses this heap for storing dynamically compiled code.

We protect both heaps from being modified by threads executing native methods, and protect the heap for dynamically compiled code from being modified by threads executing Java methods (including the runtime routines HotSpot VM provides). We permit only the threads for dynamic compilation to write to the heap for dynamically compiled code[11]. This HotSpot VM is thus secured because dynamically compiled code is protected against threads executing Java methods that have received malicious inputs that try to take control of the HotSpot VM by using bugs in the HotSpot VM to overwrite the dynamically compiled code with malicious code.

The heap protection feature we implemented works as follows.

1. At start-up, HotSpot VM creates two heap protection domains. It uses one, PD_{java}, for Java method execution, and the other, $PD_{compiler}$, for dynamic compilation. HotSpot VM sets permissions for its heap as illustrated in figure 2 by using `mprotect()`. It permits both heap protection domains to write to the Java heap and $PD_{compiler}$ to write to the heap for dynamically compiled code.

   HotSpot VM does not allow $PD_{default}$ to write to both heaps. $PD_{default}$ is a heap protection domain created by default, and HotSpot VM uses it to execute C functions. HotSpot VM permits $PD_{default}$ to read from the Java heap, so that JNI calls from C functions that cannot modify the heap can be executed without invoking `setpd()`.

2. At JNI invocation, HotSpot VM causes each thread to call `setpd()` to set its heap protection domain.

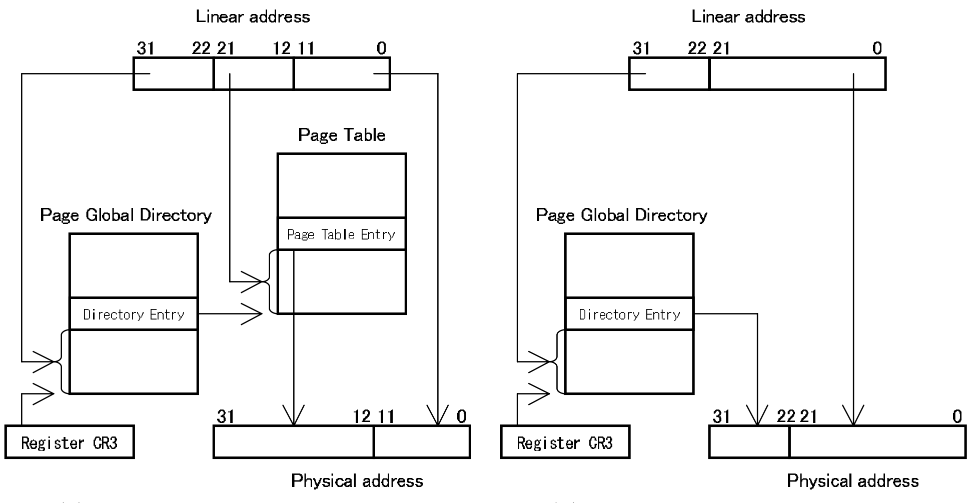
   When a thread executing a Java method calls a C function through JNI, HotSpot VM causes the thread to call `setpd()` to set its heap protection domain to $PD_{default}$, and it causes the thread to call `setpd()` again at the end of the C function execution to set its heap protection domain to PD_{java}. HotSpot VM does the opposite when a thread executing a C function calls a Java method through JNI.

3. When an invalid memory access is detected, HotSpot VM aborts immediately and dumps the core file to show where the invalid memory access has been detected.

5. IMPLEMENTATION OF SETPD()

Because `setpd()` is called twice for every JNI invocation that may modify the Java heap, its efficiency is important. Figure 3 shows a canonical implementation of `setpd()`.

[11]We temporarily permit threads running Java methods to write to the heap for dynamically compiled code when they patch the dynamically compiled code.



(a) Translation for 4-KByte pages (b) Translation for 4-MByte pages

Figure 1: Linear address translation on IA32 processors

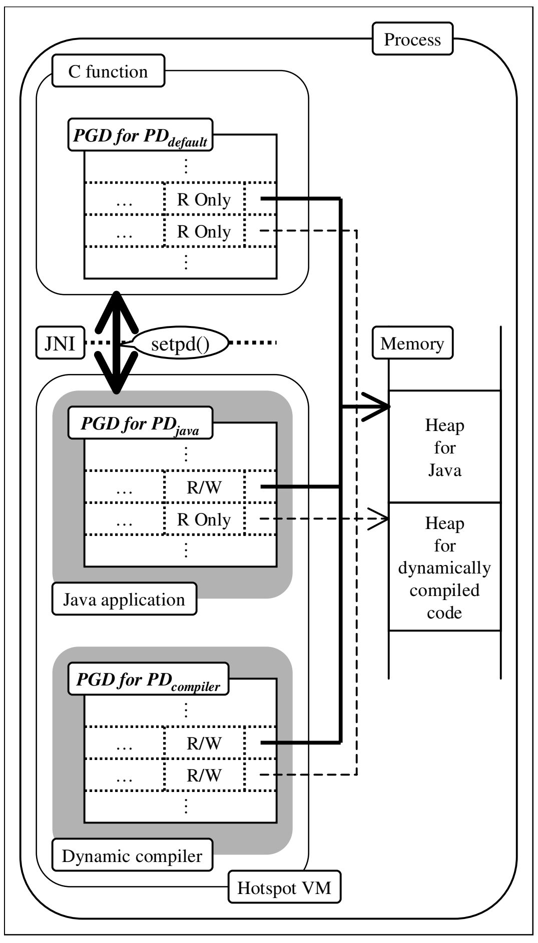


Figure 2: PGDs used by HotSpot VM

In figure 3, `setpd()` acquires a lock to traverse the protection domain at line 37, looks up `pd_struct` for its argument `new_pdid` at line 38, maintains the number of threads belonging to the new and old protection domains at lines 40–47, loads a pointers to the PGD for the new protection domain to the `CR3` register at line 48, and releases the lock at line 54.

To improve `setpd()` performance, we can refrain from supporting `deletepd()`[12] and free heap protection domains only when the process terminates. This enables the following code supporting `deletepd()` to be omitted.

Code on lines 38 and 54: The codes on these lines acquires and releases the lock for traversing the binary tree of `pd_cluster_struct` pointed to by member field `pd_tree` of `mm_struct`, but the lock is not needed if `deletepd()` is not supported. `createpd()` also modifies the binary tree as `deletepd()` does, but it can be executed in parallel with `setpd()` without the lock if it is carefully implemented: `createpd()` should initialize the PGD and `pd_cluster_struct` before connecting them to the binary tree. Otherwise, another processor executing `setpd()` in parallel with `createpd()` may refer to an uninitialized PGD or `pd_cluster_struct`, resulting in an invalid memory reference.

Code on lines 40–47: These code on these lines maintains the number of threads belonging to each protection domain so that `deletepd()` cannot delete a protection domain to which any thread belongs. There is no reason to maintain the number if `deletepd()` is not supported.

6. EVALUATION

To evaluate the runtime overhead of protecting HotSpot VM heaps, we implemented a heap protection domain on Linux (kernel 2.6.14) and heap protection for HotSpot VM

[12]Otherwise, we can implement `deletepd()` using techniques like read-copy update[**?**].

```
// Protection domain
struct pd_struct{
  pgd_t* pgd; // reference to PGD
  // number of threads belonging to
  atomic_t users;
};
// Cluster of pd_struct (a binary tree node)
struct pd_cluster_struct{
  struct pd_struct pd[PD_PER_CLUSTER];
  struct pd_cluster_struct* left;
  struct pd_cluster_struct* right;
}
// Memory manager for a process
struct mm_struct{
  ...
  // root of protection domain cluster tree
  struct pd_cluster_struct* pd_tree;
  // lock to be acquired on reference to
  the protection domain cluster tree
  rw_lock_t pd_lock;
};
// Data structure for a thread
struct task_struct{
  struct mm_struct* mm;
  ...
  // protection domain this thread belongs to
  int pdid;
};
// current thread
extern struct task_struct* current;

long setpd(unsigned int new_pdid){
  long result = current->pdid;
  if (new_pdid != current->pdid){
    struct pd_struct *new_pd;
    pgd_t *new_pgd;
    read_lock(&(current->mm->pd_lock));
    if (new_pd = find_pd(new_pdid)){
      new_pgd = new_pd->pgd;
      if (new_pdid){
        atomic_inc(&(new_pd->users));
      }
      if (current->pdid){
        struct pd_struct *old_pd =
            find_pd(current->pdid);
        atomic_dec(&(old_pd->users));
      }
      load_cr3(new_pgd);
      current->pdid = new_pdid;
    }
    else{
      result = -EINVAL;
    }
    read_unlock(&(current->mm->pd_lock));
  }
  return (result);
}
```

Figure 3: Canonical implementation of `setpd()`

Table 1: Benchmark items in SPECjvm98

Benchmark item	Description
`_201_compress`	Compression/Decompression of text files
`_202_jess`	Expert system
`_209_db`	Database
`_213_javac`	Compiler for Java source code
`_222_mpegaudio`	Decompression of MP3
`_227_mtrt`	Ray tracing using multiple threads
`_228_jack`	Parser generator

Table 2: Benchmark items in JFCMark

Benchmark item	Description
`ButtonLAF`	Look-and-feel modification
`ButtonSelect`	Button clicks
`IntFrame100`	Opening/closing internal frames
`IntFrameDrag`	Dragging an internal frame
`IntFrameLAF`	Look-and-feel modification
`IntFrameSelect`	Selecting internal frames
`BoxLayout`	BoxLayout manager test
`FlowLayout`	FlowLayout manager test
`GridLayout`	GridLayout manager test
`OverlayLayout`	OverlayLayout manager test
`SrcFineScroll`	Fine scrolling an HTML text
`SrcLoad`	Loading text files into JEditorPane
`SrcRowScroll`	Row scrolling an HTML text
`TableFineScroll`	Fine scrolling a table
`TableRowScroll`	Row scrolling a table
`TableScale`	Scaling a table
`TreeAddRem`	Adding/Removing tree nodes
`TreeExpCollDeep`	Expanding/Collapsing a deep tree
`TreeExpCollWide`	Expanding/Collapsing a wide tree

(jdk1.5.0_06) and then ran benchmarks both on the original platform (Linux and HotSpot VM) and on a platform with heap protection to compare their performance. The benchmarks were run on a PC (CPU: Intel Xeon®[13] 1.6GHz $\times$ 2, video: Radeon®[14] 9600 (resolution: 1280 $\times$ 1024, color depth: 32bit), memory: 2GByte). The benchmark suites we used were SPECjvm98, SPECjbb2000, and JFCMark version 1.0. SPECjvm98 is a suite of client-side Java applications, as summarized in table 1. SPECjbb2000 is generally used to evaluate performance of server-side business logic. JFCMark is a suite of GUI performance tests, as summarized in table 2.

The runtime options for HotSpot VM, listed below, were used with and without the heap protection.

`-Xmx -Xms` These options specify the maximum and initial heap sizes. Both were set to 512 MByte for SPECjvm98 and to 1024 MByte for SPECjbb2000. For JFCMark, the maximum heap size was set to be 40 MByte by default.

`-server` This option specifies use of the server compiler, one of HotSpot VM's two dynamic compilers. While it takes longer to compile, the compiled code is better optimized.

`-XX:+UseLargePages` This specifies that Java instances and dynamically compiled code are to be placed on large pages.[15]

The other runtime options are set to the default settings. We killed services not required for the benchmark run, such

[13]Intel Xeon and Itanium are registered trademarks of Intel Corporation and its subsidiaries in the United States and other countries.

[14]Radeon is a registered trademark of ATI Technologies Inc.

[15]Original HotSpot VM (jdk1.5.0_06 for Linux) does not place dynamically compiled code on large pages even when `-XX:+UseLargePages` is specified. We modified HotSpot VM to place dynamically compiled code on large pages when the option is specified. The modified version was used with and without heap protection.

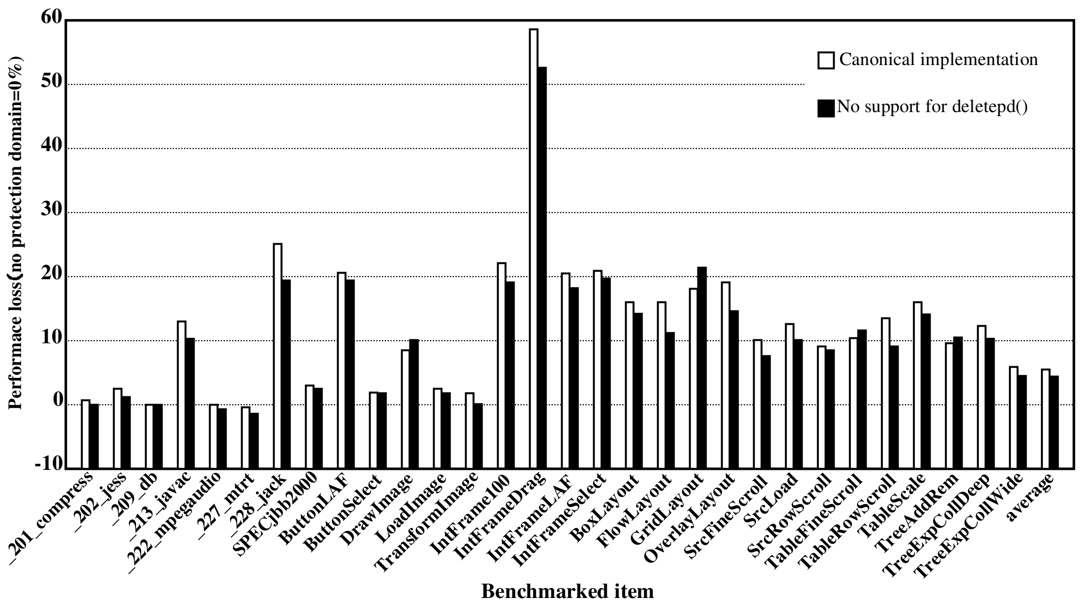


Figure 4: Runtime overhead of using the heap protection

as the networking service, as they would cause noise in the benchmark score. We executed each benchmark item 20 times and calculated the average (arithmetical mean) benchmark score to be used to calculate the performance loss due to using the heap protection. We calculated the performance loss as follows.

- If the benchmark score is the elapsed time (shorter is better), we calculated the performance loss using the following expression, where T_{on} is the elapsed time when the heap protection is used and T_{off} is the elapsed time on the original platform on which the heap protection is neither used nor implemented.

$$1 - \frac{T_{on}}{T_{off}}$$

- If the benchmark score is the number of operations processed per second (bigger is better), we calculated the performance loss using the following expression, where N_{on} is the benchmark score when the heap protection is used and N_{off} is the benchmark score on the original platform on which the heap protection is neither used nor implemented.

$$1 - \frac{N_{off}}{N_{on}}$$

The results of our evaluation are shown in figure 4. The vertical axis represents the performance loss, and the horizontal axis shows the benchmarked items. Figure 4 shows that the canonical implementation of `setpd()` resulted in an average performance loss of 5.4% (geometrical mean) and a maximum loss of 58.3%; removing support for `deletepd()` reduced the average loss to 4.4%, and the maximum loss to 52.4%. We calculated the average loss by subtracting the geometrical mean relative performance from 1, where the relative performance is $\frac{T_{on}}{T_{off}}$ or $\frac{N_{off}}{N_{on}}$.

Most JFCMark items suffered a bigger performance loss than the SPEC one because GUI operations result in JNI calls that require `setpd()`, and the `IntFrameDrag` item suffered the biggest performance loss because it makes JNI calls the most frequently, as shown in table 3. Table 3 shows the frequency of JNI calls when they are executed without using heap protection [16].

6.1 Analysis of runtime overhead

To investigate ways to reduce the runtime overhead, we analyzed the source of runtime overhead for the four benchmarked items, `_213_javac`, `_228_jack`, `IntFrame100`, and `IntFrameDrag` that imposed the most runtime overhead in each of SPECjvm98 and JFCMark. We estimated the overhead originating from two sources.

DEL Runtime overhead to support `deletepd()` came from code in the implementation of `setpd()` (lines 37, 40–47, and 54 in figure 3).

CR3 Runtime overhead to rewrite the `CR3` register.

The cost of rewriting the `CR3` register is much greater than the cost of simply executing the machine instruction that rewrites the `CR3` register because execution of the machine instruction flushes the translation look-aside buffer (TLB), which is the processor resource used to accelerate address translation. When a process uses the heap protection domain, the OS rewrites the `CR3` register when it does one of two actions.

1. Execution of `setpd()`. The code that rewrites the `CR3` register is at line 48 in figure 3.

[16]Only JNI calls that invoke setpd() when executed using heap protection were counted. JNI calls from C functions that cannot modify the Java heap do not invoke `setpd()`, as described in section 4.

Table 3: Frequency of JNI calls that invoke `setpd()`

Benchmarked item	Frequency (times/sec)
`_201_compress`	473
`_202_jess`	7491
`_209_db`	1440
`_213_javac`	34429
`_222_mpegaudio`	754
`_227_mtrt`	3930
`_228_jack`	45434
`ButtonLAF`	57514
`ButtonSelect`	5791
`DrawImage`	40253
`LoadImage`	5912
`TransformImage`	380
`IntFrame100`	74408
`IntFrameDrag`	202444
`IntFrameLAF`	66039
`IntFrameSelect`	75157
`BoxLayout`	45094
`FlowLayout`	56277
`GridLayout`	56291
`OverlayLayout`	64047
`SrcFineScroll`	35385
`SrcLoad`	33217
`SrcRowScroll`	26445
`TableFineScroll`	34967
`TableRowScroll`	32739
`TableScale`	49793
`TreeAddRem`	21514
`TreeExpCollDeep`	37797
`TreeExpCollWide`	15667

2. Context switch from a thread that belongs to a heap protection domain to a thread that belongs to another protection domain.

The former accounted for most of the runtime overhead associated with the four benchmark items because these benchmark items call `setpd()` very frequently, as table 3 shows, while the OS does not switch contexts so often.

Using the canonical implementation of `setpd()` shown in figure 3, we estimated the runtime overhead from `DEL` and `CR3` as follows.

1. For each of the benchmark items, we measured the execution time for four cases.

   $T_{original}$ Neither the heap protection domain nor the heap protection feature was implemented.

   $T_{protect}$ The heap protection domain was implemented and used to protect the HotSpot VM heap.

   T_{noDEL} The heap protection domain was implemented and used to protect the HotSpot VM heap, but the code supporting `deletepd()` was omitted.

   T_{noCR3} The heap protection domain was implemented and used to protect the heap for HotSpot VM, but the code to rewrite the `CR3` register for the protection domain was omitted.

2. The share of total runtime overhead for `DEL` and CR3 when using the heap protection was calculated using following expressions.

Table 4: Source of performance loss and execu cost of`setpd()`

Benchmarked item	Performance loss(%)			Executic cost(μs
	O_{DEL}	O_{CR3}	others	
`_213_javac`	19.8	53.2	27.0	1.9
`_228_jack`	22.8	60.8	16.4	2.8
`IntFrame100`	17.4	51.0	31.6	1.9
`IntFrameDrag`	22.2	56.8	21.0	3.3

$$O_{DEL} = \frac{T_{protect} - T_{noDEL}}{T_{protect} - T_{original}}$$
$$O_{CR3} = \frac{T_{protect} - T_{noCR3}}{T_{protect} - T_{original}}$$

Because the execution time was not consistent bet the runs, the estimation accuracy was not good, how the estimations should be roughly approximate. The r of the estimation is shown in table 4.

As shown in table 4, O_{DEL} and O_{CR3} differed amon four benchmarked items, possibly due to inconsistenc runtime overhead to rewrite the `CR3` register. The ru overhead to rewrite the `CR3` register includes the ru overhead due to TLB flush, which is affected by the me access pattern and flush timing. The average cost of flu TLB drops when flushing is done continually and a part of TLB is reloaded between flushing. The cost t `setpd()` measured using a program that continually `setpd()` was 1.17μs, which is smaller than the cost s in table 4 [17]. This is because, in practical applications, of TLB is reloaded between flushes.

As shown in table 4, over half the runtime overhead from `CR3`, so we must reduce this overhead to improve formance. However, it is not easy to reduce the overhe improving the implementation of `setpd()`.

- Shinagawa[25] used segmentation provided by processors to avoid TLB flushing on a protectio main switch. But this method may not be approp for server-side Java applications that consume bytes of heap, because it reduces the available lo address space as the number of protection domai the process grows. However, Shinagawa's imple tation can be practical for PowerPC processor Both IA32 and PowerPC processors implemen tual memory using segmentation and paging. first translate each logical address into an inte diate address using segmentation, and then tra the intermediate address into a physical addres ing paging. While the intermediate (linear) ad space is 32-bit in IA32 processors, it is 52-bit i bit PowerPC processors. Shinagawa's implement reduces the available logical address space becau divides the 32-bit linear address space of IA32 cessors among the protection domains and then the logical address space to the fragment of the l address space, but PowerPC processors can divi large-enough (52-bit) intermediate address space 32-bit pieces and give one to each protection do

[17]We estimated the cost of `setpd()` for each benchm item by simply dividing total overhead of using the protection ($T_{protect} - T_{original}$) by the execution cou `setpd()`. This estimation neglects runtime overhead t not due to `setpd()`.

In this case, each protection domain can fully access the 32-bit logical address space.

- Palladium[5] also used segmentation to implement a memory protection scheme like our heap protection domain. Palladium does not suffer the problem that Shinagawa's implementation does. It classifies programs into core programs and their extension programs. It prevents an invalid memory access by the extension programs to the memory area for their core program. It can protect memory more efficiently than our heap protection domain does, because it requires neither TLB flushing nor a system call on a protection domain switch. However, Palladium cannot prevent an extension program from invalidly accessing the memory for another extension program, and it cannot be implemented on platforms without segmentation. In contrast, our heap protection domain allows each subprogram to have its own protected heap and can be portably implemented.

- Processor support for the protection domain can reduce the frequency of TLB flushing. The processor can use reserved (unused) bits in the directory entry to add access control bits for multiple protection domains in one directory entry. This reduces not only the number of PGDs but also the frequency of TLB flushes, because a TLB flush is not needed with `setpd()` if old and new protection domains to which the thread belongs share a PGD. Unfortunately, there is not yet a processor that supports this capability.

  Another processor support for the protection domain is a tagged TLB[1, 14, 29]. A tagged TLB is a TLB that can hold address transformation information for multiple address spaces. With this feature, there is no need to flush the TLB on an address space switch, and this enables an efficient implementation of a protection domain, which is implemented as an address space[30].

The runtime overhead due to `DEL` can be eliminated by not supporting `deletepd()` as described in section 5. There is little room for runtime overhead from other than `CR3` or `DEL` to be reduced, as most of it is due to code executed in common for every system call, such as interrupt instructions.

Runtime overhead can also be reduced by eliminating `setpd()` calls. It is possible to omit most `setpd()` calls in SPECjvm98 and SPECjbb2000 if the HotSpot VM eliminates calls to `setpd()` in JNI calls to the native methods provided by the JRE. It may not be important for HotSpot VM to call `setpd()` in JNI calls to the native methods provided by the JRE, because HotSpot VM and the native method are together in one program product, the JRE, and engineers analyzing crashed Java applications generally ask the JRE developer to debug the problem whether the bug is in the JVM or in the native method provided by the JRE. This eliminates most `setpd()` calls for pure Java applications, such as SPECjvm98, SPECjbb2000 and JFCMark, but this is of no help for Java applications that frequently call their own native methods.

7. CONCLUSION

We have described a feature that protects the heap for Java virtual machines using the heap protection domain. We modified Linux and HotSpot VM to implement this heap protection and evaluated the performance loss using three benchmarks suites, SPECjvm98, SPECjbb2000, and JFCMark. The evaluation showed that the average performance loss was 4.4% and the maximum loss was 52.4%.

8. ACKNOWLEDGMENTS

We would like to thank Shingo Takada for his helpful comments. We also want to thank the referees of this paper for their useful suggestions.

9. REFERENCES

[1] The Alpha Architecture Committee. *Alpha AXP architecture reference manual, third edition.* Digital Press, Boston, Massachusetts, 1998.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Fourth Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2005.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 290–301. June 1994.

[4] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer System*, 12(4):271–307, November 1994.

[5] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the seventeenth ACM Symposium on Operating System Principles*, pages 140–153. December 1999.

[6] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceedings of the fifth ACM Symposium on Operating System Principles*, pages 141–160. November 1975.

[7] G. Czajkowski, L. Danyés, and N. Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 155–177, June 2002.

[8] G. Czajkowski, L. Danyés, and M. Wolczko. Automated and portable native code isolation. In *Proceedings of 12th IEEE International Symposium on Software Reliability Engineering*, pages 298–307. November 2001.

[9] C. Demetrescu and I. Finocchi. A portable virtual machine for program debugging and directing. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1524–1530. March 2004.

[10] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[11] Excelsior, Limited Liability Company. JFCMark, 2003. *http://www.excelsior-usa.com/jfcmark.html*

[12] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 341–348. May 1981.

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 2006. http://www.intel.com/design/Pentium4/documentation.htm

[14] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, 2006. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm

[15] International Business Machines Corporation. *ILE Concepts*. International Business Machines Corporation, 2002. http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/c4156066.pdf

[16] International Business Machines Corporation. *Java and WebSphere Performance on IBM iSeries Servers*. International Business Machines Corporation, 2002. http://www.redbooks.ibm.com/redbooks/pdfs/sg246256.pdf,

[17] International Business Machines Corporation. *PowerPC Architecture Book*. International Business Machines Corporation, 2003. http://www-128.ibm.com/developerworks/eserver/articles/archguide.html

[18] R. W.M. Jones and P. H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated and Algorithmic Debugging*, pages 13–26. May 1997.

[19] T. Koju, S. Takada, and N. Doi. An Efficient Directing Platform Compatible with Existing Development Environments *IPSJ Journal*, 46(12):3040–3053, December 2005.

[20] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the fifth ACM Symposium on Operating System Principles*, pages 132–140. November 1975.

[21] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[22] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, Washington, 2001.

[23] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139. January 1997.

[24] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Proceedings of Java Virtual Machine Research and Technology Symposium*, pages 1–12. April 2001.

[25] T. Shinagawa, K. Kono, and T. Masuda. Flexible and efficient sandboxing based on fine-grained protection domains. In *Proceedings of the International Symposium on Software Security*, pages 172–184, February 2003.

[26] R. Sosič. Dynascope: a tool for program directing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 12–21. July 1992.

[27] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks, 1998. *http://www.spec.org/osg/jvm98/*

[28] Standard Performance Evaluation Corporation. SPECjbb2000, 2000. *http://www.spec.org/osg/jbb2000/*

[29] SPARC International, Incorporated. *SPARC Architecture Manual, Version 9*. SPARC International, Incorporated, 2000. http://developer.sun.com/solaris/articles/sparcv9.html

[30] M. Takahashi, K. Kono, and T, Masuda. Efficient kernel support of fine-grained protection domains for mobile code, In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 64–73. May 1999.

[31] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* , pages 304–316. October 2002.

[32] W. Wulf, R. Levin, and C. Pierson. Overview of the hydra operating system development. In *Proceedings of the fifth ACM Symposium on Operating System Principles*, pages 122–131. November 1975.

A Framework for Unified Resource Management in Java

Derek A. Park
The University of Mississippi
Global Technical Systems
9 Industrial Park Drive, Suite 105
Oxford, MS 38655 USA
+1 (662) 236-6200
dpark@olemiss.edu

Stephen V. Rice
The University of Mississippi
P.O. Box 1848
University, MS 38677-1848 USA
+1 (662) 915-5359
rice@cs.olemiss.edu

ABSTRACT

Although Java automates the deallocation of memory through garbage collection, a Java program must explicitly free other resources, such as sockets and database connections, to prevent resource leaks. Correct and complete resource deallocation may be complex, requiring nested *try-catch-finally* blocks. The Framework for Unified Resource Management (Furm) is a Java library designed with the goal of simplifying resource management in single- and multi-threaded programs. Allocated resources are represented by nodes in "resource trees" and resource dependencies are modeled as parent-child relationships. A hierarchy of resources is easily and correctly deallocated by a single call of Furm's *release* method. Resource trees permit "resource reflection" in which a program monitors its own resource usage. A special *WatchDog* object detects when a thread dies and takes care of releasing its resources. Furm reduces code complexity while guaranteeing resource cleanup in the face of exceptions and thread termination.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming. D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming.*

General Terms

Languages, Reliability, Management.

Keywords

Resource monitoring and deallocation, Exception handling.

1. INTRODUCTION

Resource management is a concern which reaches all programs. At the operating system level, heap and stack memory are granted to programs and system handles are allocated. At the program level, memory is allocated and freed; sockets and files are opened and closed. Even the most trivial program performs resource management, or has resource management performed on its behalf. An empty program still has a stack which the operating system must free.

Allocating resources is simple enough. The complexities arise from the need to deallocate resources. Memory must be freed. Sockets must be closed. File handles must be released. When resources are not freed, resource leaks inevitably occur. The more complex a program becomes, the more likely such a leak is to occur. In Java and other exception-handling languages, resource deallocation is made even more difficult by the complexities of exception-handling.

It has been noted that the exception-handling model is in fact more complex and difficult to use correctly than the more traditional error-code model, in which functions signal errors by returning special values [14][5][1]. The difficulty in correctly handling exceptions often leads to incorrect handling. When concurrency issues are introduced, the problem grows further.

1.1 Exception-Handling Complexity

Why does exception handling complicate resource management? It makes the code flow unpredictable. In a language like C, it is possible to allocate resources, confirm successful allocation, use those resources, and then free them. The flow of code is very straightforward. In Java however, virtually any statement may throw an exception. It is therefore impossible to simply allocate resources and then free them. If an exception is thrown while allocating, using, or freeing resources, any resources allocated but not yet freed will be leaked when the exception unwinds the stack, moving the flow of control in a potentially unexpected pattern.

As an illustrative example, take the following code which attempts to close three resources, a socket, a database connection, and an input stream:

```
01 socket.close();
02 connection.close();
03 input.close();
```

In a language without exceptions, each of these resources would be guaranteed cleanup. In Java, however, if the socket fails to close in line 1, it will throw an exception, and the other resources will never be released.

Because guaranteed cleanup is necessary, Java has the *try-finally* block. Java guarantees that if the flow of code reaches a *try* block, the associated *finally* block is guaranteed to be executed, even if an exception is thrown inside the *try* block. Using *try-finally* blocks, we might revise our code as follows:

```
01 try {
02   ...
03 } finally {
04   try {
05     socket.close();
06   } finally {
07     try {
08       connection.close();
09     } finally {
10       input.close();
11     }
12   }
13 }
```

While this code will now correctly release our three resources, even in exceptional cases, it has ballooned from a simple three lines to a much more bulky thirteen. The extra overhead is necessary to guarantee that each resource will be released.

Note that there are three nested *try-finally* blocks to release three resources. If there were four resources, we would require an additional nested *try-finally* block. In fact, correctly releasing *N* resources requires *N try-finally* blocks [17].

Finally, it is worth pointing out that even catching exceptions will not necessarily guarantee correct resource cleanup. The Java language has two kinds of exceptions: checked and unchecked. Java requires that a method either catch all possible checked exceptions or explicitly declare that it may throw them. Unchecked exceptions, the subclasses of *Error* and *RuntimeException*, are not required to be declared or caught. Any method may throw an unchecked exception at any time. This means that catching only declared exceptions is not sufficient to guarantee cleanup, as illustrated in the following code:

```
01 try {
02   socket.close();
03 } catch (IOException exc) {
04 }
05 connection.close();
```

The *Socket.close()* method only declares that it might throw *IOException*, so it might seem reasonable to assume that catching *IOException* in line 3 will be enough to guarantee that the flow of control will reach line 5. If, however, line 2 were to throw an unchecked exception, such as *NullPointerException*, then it would go uncaught, and so the database connection would remain open.

1.2 A Motivating Example

While exception-handling clearly imposes a burden when releasing resources, it is worth asking whether that burden is really a problem. Figure 1-1 is an example taken from *The Java Tutorial* [4]. The code is intended to show, along with use of sockets, that resources opened must be also closed. However, this code has some of the same flaws already discussed.

If any unchecked exceptions are thrown in lines 8-14, they will not be caught by the catch blocks on lines 15 and 17. The rest of the method would not execute, leaking any of the successfully opened resources. Another likely issue is that if the connection fails while writing to or reading from the streams (lines 27-31), an exception will be thrown. The socket and streams are guaranteed to be leaked if this occurs. No *catch* or *finally* blocks are present to guard against this.

1.3 Garbage Collection and Finalization

Because memory is the most commonly allocated resource, it follows that it is also the most commonly mismanaged resource. The concept of automatic garbage collection was introduced to eliminate the possibility of memory leaks. Garbage collection has done an admirable job of eliminating memory leaks. It has often been asserted that programmers using garbage-collected languages are more productive, specifically because the need to deal with memory deallocation is absent [13].

Unfortunately, no such solution has been presented to eliminate the complexities of the more general case of resource management. C++ introduced the awkwardly-named Resource Acquisition Is Initialization (RAII) paradigm, which involves using automatically-destructed, scope-bound objects to release resources [1][15]. However, such a feature is not available in Java, because automatically-destructed objects are not available. In Java, all non-memory resource deallocation must be done manually [4][3].

Java's analog for C++ destructors is finalization. Rather than deleting an object manually and implicitly calling its destructor (as in C++), when an object in Java is garbage collected, its *finalize()* method is called [8]. In theory, this is an excellent time to free resources. Since the object is being garbage collected, its resources are clearly no longer needed. Unfortunately, this model is flawed. The Java specification does not guarantee that objects will be garbage collected quickly. It is accepted, however, that most non-memory resources need to be freed quickly. Examples include network sockets, file handles, and database connections

```
01 public static void main(String[] args)
02                    throws IOException {
03   Socket echoSocket = null;
04   PrintWriter out = null;
05   BufferedReader in = null;
06
07   try {
08     echoSocket = new Socket("taranis", 7);
09     out = new PrintWriter(
10             echoSocket.getOutputStream(),
11             true);
12     in =  new BufferedReader(
13             new InputStreamReader(
14               echoSocket.getInputStream()));
15   } catch (UnknownHostException e) {
16     // ...
17   } catch (IOException e) {
18     // ...
19   }
20
21   BufferedReader stdIn =
22           new BufferedReader(
23             new InputStreamReader(
24               System.in));
25
26   String str;
27   while ((str = stdIn.readLine()) != null) {
28     out.println(str);
29     System.out.println("echo: " +
30                      in.readLine());
31   }
32
33   out.close();
34   in.close();
35   stdIn.close();
36   echoSocket.close();
37 }
```

Figure 1-1. An example from *The Java Tutorial* [4]

[4]. The fact that the Java framework has no way of explicitly releasing memory[1], but has numerous ways of closing other resources explicitly[2], supports this view.

The fact that Java's finalizers are not guaranteed to be run in a timely fashion would seem to limit their usefulness. Even more limiting, however, is that finalizers are in fact not guaranteed to be run at all [9][3]. An object's finalizer is only guaranteed to be run if the object is garbage collected, and garbage collection is not guaranteed to happen [4]. It is entirely possible for a program to exit without any of its objects being garbage collected or finalized. Java does provide a way to force objects to be finalized on exit[3], but the functionality has been deprecated since Java 1.2, as it has been deemed unsafe. Without any accepted way to guarantee finalization, exactly how finalization is at all useful becomes a valid question.

Because finalization's usefulness is limited, Java code must manually release all scarce resources. Java does not make this job simple. While releasing a resource typically boils down to a single method call, the exact call is not standardized across the Java libraries. Most resources provide a *close()* method, including *Socket*, *InputStream*, *Connection*, etc. However, a few resources stray from that pattern, such as the AWT *Window* class, which provides the *dispose()* method. Even those classes which do provide a *close()* method are not truly consistent, because Java does not have a single, unified interface for closing resources. The closest the library comes to a unified interface is *Closeable*. The *Closeable* interface is implemented by all resources in the *java.io* package. However, outside the *java.io* package, few resources implement the *Closeable* interface, and so many resources cannot be treated as instances of *Closeable*. The net effect of what are in fact fairly minor inconsistencies is that there is no straightforward way to release all resources in an identical manner, which complicates any attempt at automated resource deallocation.

1.4 Releasing Resources

Dependencies are inherent in the very nature of resources. A JDBC *PreparedStatement* object has no logical meaning in the absence of an associated *Connection* object. Similarly, the input and output streams associated with a network socket have no further meaning once the socket they are built upon closes. Upon inspection, it becomes clear that there are many such dependencies, and they are applicable to all sorts of resources.

While at first it might seem logical that when a resource closes, any resources which depend on that resource will also close automatically, this cannot be safely assumed in the general case. Any behavior which is not defined is undefined, axiomatically. Undefined behavior cannot be trusted to act consistently, and the Java documentation is usually quiet when it comes to automatic resource closure. One might assume that when a database connection closes, any statements tied to that connection will also close. However, the Java 1.5 API documentation makes no such claim [16].[4] Accordingly, it falls on the programmer to explicitly close any open statements along with the database connection. Likewise, the API documentation does not state that closing a socket automatically closes its associated streams, which leaves this task to the programmer. In fact, the recommended method of closing multiple dependent resources is to close them manually, one at a time [4].

Whenever the documentation is silent about whether a resource closing will cause any dependent resources to close, it is best to assume nothing, and treat the dependent resources as potentially unusable. In such a case, the only logical thing to do with the dependent resources is to close them as well. When a socket is closed, the programmer should immediately close the associated input and output streams. In fact, this should be done before the socket is closed [4]. It is interesting to note that in the Java 1.5 implementation provided by Sun, closing a stream associated with a socket actually will close the socket [10], but this behavior is unreliable. It is not guaranteed between Java versions – in fact it is a recent change – and it is undocumented, which means other Java implementations may not exhibit the same behavior.

1.5 Related Work

Finalizers were introduced into Java as a way to guarantee cleanup [8], but it has been shown that finalizers fail in their task. Stack-bound objects with destructors have been given thorough attention in C++ [15], but such approaches are not applicable to Java, due to its lack of stack-bound objects.

C# and .NET introduced the *IDisposable* interface, which is used throughout the .NET framework. This provides the unifying interface missing from Java. C# also introduced the *using* keyword (coupled with the *IDisposable* interface) to simplify resource management [7]. The *using* construct eliminates the need for *try-finally* blocks when releasing resources, and even automates the resource release. One drawback to applying this technique to Java is that it requires language-level support, and so requires a language extension. The second issue is that it requires *IDisposable*, or an equivalent unified interface, to work.

Czajkowski provided a thorough treatment of the area of resource allocation and usage monitoring [6]. However, the methods presented are not applicable to releasing resources. The methods do not eliminate the need for nested *try-finally* blocks when releasing resources.

Weimer delved deeply into the issue of releasing resources [17] using compensations, arbitrary pieces of code that are executed to restore invariants. The framework presented by Weimer stores these compensations on a stack, and the entire stack is typically executed at once. However, compensations require changes to the Java language itself, to add closures and an extended syntax. Compensation stacks are also less flexible than the resource tree model we will introduce in the next section.

[1] Java does provide a way to request that the garbage collector be immediately run, but this practice is discouraged, and is not guaranteed to free any memory, only to make an attempt.

[2] Various calls are used to close resources, including *Closeable.close()*, *Window.dispose()*, *Connection.close()*, etc.

[3] *Runtime.runFinalizersOnExit()*

[4] The JDBC spec states that closing a *Connection* object closes the associated *Statement* objects, although this is not reflected in the more general Java 1.5 API documentation. The JDBC spec states, however, that *Statement* and *ResultSet* objects should be explicitly closed as soon as possible, implying that the automatic closure is a fallback mechanism only.

2. FRAMEWORK

The Framework for Unified Resource Management (Furm) represents dependencies among resources using trees [12]. In a resource tree, given a parent resource P and a child resource C, C is dependent on P. That is, once P is closed, C has no further use. By extension, given any two resources X and Y where X is an ancestor of Y, Y is dependent on X, either directly or indirectly. All resources are therefore dependent on the tree's root resource.

This resource tree abstraction is primarily useful in two ways. First, the tree provides a unified way to manage resources. This fills the gap left by Java's *Closeable* interface. The resource tree provides a consistent way to view resources, and makes it possible for all resources to be released in an identical manner.

Second, the resource tree materializes resource dependencies in an explicit fashion. While an input stream might be associated with a socket, it may not be evident to any code using the stream that this is the case. However, if the socket and its streams are part of the same resource tree, the relationship becomes clear. The connection in the tree ties a parent and child together logically and simply.

Furm models two types of dependencies: substantive dependencies and logical dependencies. Figures 2-1 and 2-2 illustrate substantive and logical dependencies, respectively. Substantive dependencies are those dependencies that might be described as concrete, physical, or real dependencies. These are forced dependencies, or dependencies in which the child resource truly has no meaning in the absence of the parent resource. The resource dependency examples used so far have all been substantive dependencies.

Logical dependencies are semantic dependencies, or groupings of resources. For example, the input and output streams used by a block of code may not have any true, enforced dependencies, but they may still be logically dependent, in that without both, execution cannot successfully proceed. It is therefore useful to represent them as siblings with a common parent. Neither the input stream nor the output stream has a substantive dependency on the other, but they are logically connected. Having a common parent materializes this relationship. In logical dependencies, the parent will often be a pseudo-resource, i.e., the parent represents a group of resources, and does not itself represent a particular resource. It exists to support the semantic relationship of its children. A common use of logical dependencies is to group all the resources of a segment of a program. This might be a code block, a specific object, or a thread of execution. Figure 2-3 shows a resource group associated with a thread, which contains a database connection and a socket, each with its own dependencies.

It has been shown that the resource tree can represent dependencies among resources. In addition, Furm propagates resource releases from ancestors to descendents. If a resource in the tree is released, then all of its children will be released, and in turn the children's children, etc. Rather than releasing a number of individual resources, only the parent resource needs to be explicitly released. Entire subtrees of resources can be released with a single call. This has the immediate effect that cleanup code, with its large code overhead, is greatly reduced.

2.1 Using Furm

Here we present a simple method *fetch()*, with the job of connecting to a server, sending a message, and returning the response to the caller. (See Figure 2-4.) In the event of failure, the response is null.

This method is implemented correctly, with respect to resource management. The three resources used are a *Socket*, an *InputStream*, and an *OutputStream*. The *ByteArrayOutputStream* is not considered a resource here, because it does not need to be closed.

While the task of this code is simple, the decision to correctly close resources and handle all checked exceptions – as opposed to discarding them or allowing them to propagate to the caller – has caused massive code expansion. If exceptions could be safely ignored, this method could be reduced from 44 lines to only 18. Note especially how the nested *try-finally* blocks turn cleanup, a conceptually minor detail, into the most dominant detail.

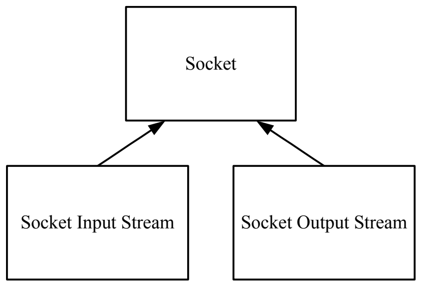


Figure 2-1. Substantive dependencies

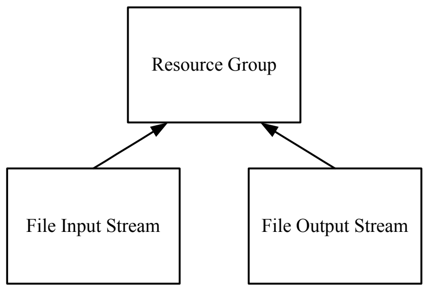


Figure 2-2. Logical dependencies

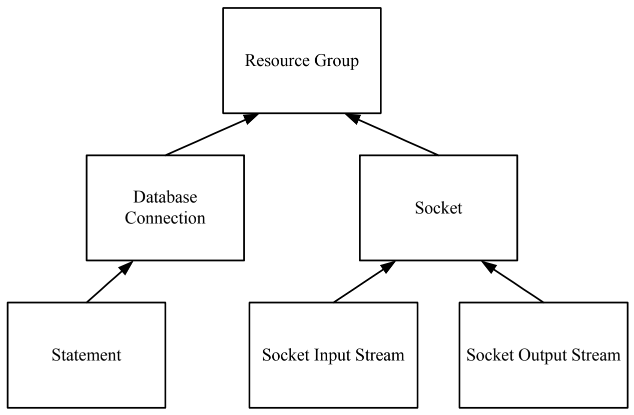


Figure 2-3. An example resource tree

By representing resources as trees, and releasing entire trees at once, this method can be simplified, as shown in Figure 2-5. The *try-catch-finally* complexity has been reduced to a single *try-catch-finally* block. The previous *close()* calls have been replaced by a single *release()* call, on line 27. Exceptions during the resource release phase are now passed to the *ExceptionLogger*, described in Section 2.3, to avoid cluttering up the code. Additionally, it is no longer necessary to declare most of the resources outside the *try* block (e.g., the *Socket*, *InputStream*, and *OutputStream*). Because the resources are accessible through the tree, they do not need to be directly accessible by the *finally* block. This allows more variables to be kept in smaller scopes, where they are relevant. By using Furm, the number of lines has been significantly cut. More importantly, communicating with the server is now the focus of the code. The resource management code has been pushed into the background, where it belongs.

In this example, a *ResourceGroup* is the root of the tree, and the *ManagedSocket* is its only child. When *getInputStream()* and *getOutputStream()* are called, the *ManagedSocket* automatically creates a *ManagedInputStream* and a *ManagedOutputStream* as its children. Figure 2-6 shows the tree immediately before *release()* is called.

```
01 static byte [] fetch(InetAddress addr,
02                      int port,
03                      byte [] msg) {
04   Socket s = null;
05   InputStream in = null;
06   OutputStream out = null;
07   ByteArrayOutputStream response =
08              new ByteArrayOutputStream();
09   boolean success = false;
10   try {
11     s = new Socket(addr,port);
12     out = s.getOutputStream();
13     out.write(msg);
14     out.flush();
15     in = s.getInputStream();
16     int i;
17     while ((i = in.read()) != -1) {
18       response.write(i);
19     }
20     success = true;
21   } catch (IOException exc) {
22     // log exception
23   } finally {
24     try {
25       if (out != null) out.close();
26     } catch (IOException exc) {
27       // log exception
28     } finally {
29       try {
30         if (in != null) in.close();
31       } catch (IOException exc) {
32         // log exception
33       } finally {
34         try {
35           if (s != null) s.close();
36         } catch (IOException exc) {
37           // log exception
38         }
39       }
40     }
41   }
42   if (!success) return null;
43   return response.toByteArray();
44 }
```

Figure 2-4. Connecting to a server to send a message

For another example, let us borrow a method from Apache Tomcat, an open source Java Servlet container, produced by the Apache Software Foundation. Tomcat is the official Reference Implementation for the Java Servlet and JavaServer Pages technologies [2].

The *copy()* method in Figure 2-7 is from the *StandardContext* class, part of the Java engine (Catalina) built into Tomcat. Its task is to copy from one file to another. A significant amount of effort has gone into handling resources in exceptional situations in this code. All resource releases are nicely wrapped in *try-catch* blocks, which are in turn in a *finally* block. The method is well thought out, carefully coded, and flawed. If the *close()* operation on line 19 throws an *Error*, then the *close()* operation on line 22 will never occur, despite all the careful exception handling. This is unlikely, but not impossible. This may have slipped by the developers because the cleanup code is difficult to follow.

Note that the *close()* calls are duplicated. Lines 13 and 14 close the resources in the normal conditions, while lines 19 and 22 close the resources in exceptional conditions. Such duplication leads to less maintainable code. Note also that exceptions are completely discarded, losing potentially useful information.

Figure 2-8 displays the code rewritten to use Furm. This code correctly closes resources in both normal and exceptional situations. The basic functionality is retained. However, the cleanup code is completely different. Instead of nested *try-catch* blocks, a single *try-catch-finally* block suffices. Rather than two different *close()* calls (or four, counting the duplicates), a single *release()* call is responsible for releasing the input and output streams correctly in both normal and exceptional cases. The return value of the *release()* call is checked on line 19 to determine whether the cleanup was successful. A return value of

```
01 static byte [] fetchAgain(InetAddress addr,
02                           int port,
04                           byte [] msg) {
05   ResourceGroup tree = new ResourceGroup(
06                new ExceptionLogger());
07   ByteArrayOutputStream response =
08                new ByteArrayOutputStream();
09   boolean success = false;
11   try {
12     Socket s = new ManagedSocket(
13                     addr,port,tree);
14     OutputStream out = s.getOutputStream();
15     out.write(msg)
16     out.flush();
17     InputStream in = s.getInputStream();
18     int i;
19     while ((i = in.read()) != -1) {
20       response.write(i);
21     }
22     success = true;
23   } catch (IOException exc) {
24     tree.getListener().
25                exceptionThrown(null,exc);
26   } finally {
27     tree.release();
28   }
29   if (!success) return null;
30   return response.toByteArray();
31 }
```

Figure 2-5. Connecting to a server using Furm

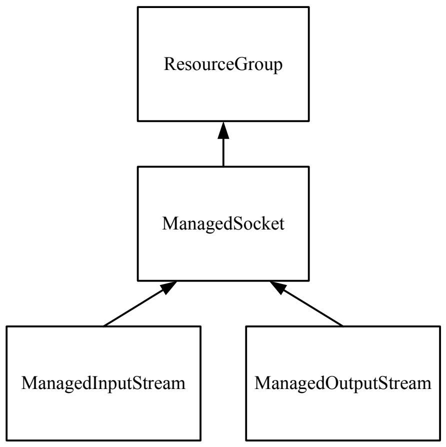


Figure 2-6. A potential resource tree associated with a Socket

zero indicates that no exceptions occurred.

Among the most obvious differences is the reduction in cleanup code. The release and exception-handling code has been reduced by almost half. The nested *try-catch* blocks are no longer needed because that functionality is now encapsulated in the *release()* call. Additionally, exceptions are no longer discarded silently. Instead they are passed to an *ExceptionListener*. The overall method is shorter as well, even with the overhead of creating the *ResourceGroup*. Compared to the original, use of Furm reduced the amount of code, eliminated the resource cleanup error, and minimized the loss of information related to exceptions.

2.2 Creating Managed Resources

Adding new resources to Furm is straightforward. The most difficult issue with using Furm is that each type of resource to be added to the tree must be wrapped in a new class to produce the equivalent managed resource. For example, to add an *InputStream* to the tree, a new wrapper class, *ManagedInputStream*, must be defined. Additionally, each managed resource must define its own subclass of *ResourceNode*, described in Section 2.3, to override the *cleanup()* method. In practice, these requirements pose little problem.

Figure 2-9 shows one possible implementation of the *ManagedInputStream* class. This class extends *FilterInputStream* to wrap an underlying *InputStream*. *ManagedInputStream* uses an anonymous inner class [8] to subclass *ResourceNode* and override its *cleanup()* method. Use of an anonymous inner class is not necessary. A named class would be sufficient, but less succinct. The *close()* method is also overridden to call the *release()* method of *ResourceNode*. This will automatically release any child resources and call the *cleanup()* method of the *ResourceNode* subclass, which in turn will call *ManagedInputStream*'s superclass *close()* method.

ManagedInputStream exposes its *ResourceNode* instance variable publicly to expose the *ResourceNode* functionality. This is the recommended practice. Because forwarding all relevant calls to the underlying *ResourceNode* would be a tedious task, the choice was made to simply expose the *ResourceNode* directly, rather than hide it. In practice, this means little except that to release the resource, the correct call is *stream.node.release()* rather than *stream.release()*. For the sake of consistency, the name *node* is recommended, but not mandated by Furm. The *node* variable should be declared *final* to protect it from modification.

It is recommended that managed resource classes *extend* (or *implement*) the class they are intended to replace. In this example, *ManagedInputStream* extends *FilterInputStream* (so that it indirectly implements *InputStream*). This allows *ManagedInputStream* to be passed to methods and constructors which expect an *InputStream*. *ManagedInputStream* thus becomes a drop-in replacement.

In cases where the original resource is a class, the constructors of the original resource class will need to be duplicated in the managed subclass. In cases where a resource has a large number of constructors, this could be tedious. However, a tool could automate this task. In cases where the original resource is an

```
private boolean copy(File src, File dest) {
  FileInputStream is = null;
  FileOutputStream os = null;
  try {
    is = new FileInputStream(src);
    os = new FileOutputStream(dest);
    byte[] buf = new byte[4096];
    while (true) {
      int len = is.read(buf);
      if (len < 0) break;
      os.write(buf, 0, len);
    }
    is.close();
    os.close();
  } catch (IOException e) {
    return false;
  } finally {
    try {
      if (is != null) is.close();
    } catch (Exception e) {/*Ignore*/}
    try {
      if (os != null) os.close();
    } catch (Exception e) {/*Ignore*/}
  }
  return true;
}
```

Figure 2-7. The *copy()* method from Tomcat's Java engine

```
private boolean copy(File src, File dest) {
  ResourceGroup root = new ResourceGroup(…);
  boolean success = false;
  try {
    InputStream is =
      new ManagedFileInputStream(src,root);
    OutputStream os =
      new ManagedFileOutputStream(dest,root);
    byte[] buf = new byte[4096];
    int len;
    while ((len = is.read(buf)) >= 0) {
      os.write(buf, 0, len);
    }
    success = true;
  } catch (IOException exc) {
    root.getListener().
              exceptionThrown(null, exc);
  } finally {
    if (root.release() != 0) success = false;
  }
  return success;
}
```

Figure 2-8. A version of the *copy()* method using Furm

```
public class ManagedInputStream
                  extends FilterInputStream {
  public final ResourceNode node;
  public ManagedInputStream(
            InputStream input,
            ResourceNode parent)
         throws NullPointerException {
    super(input);
    resource = new ResourceNode (this,
                                parent) {
                  public void cleanup()
                           throws Exception {
                    ManagedInputStream
                           .super.close();
                  }
                };
  }
  public void close() {
    node.release();
  }
}
```

Figure 2-9. A *ManagedInputStream* implementation

interface rather than a class, method calls must be forwarded. Again, this could largely be automated. For a few resources, simple wrapper classes, such as *FilterInputStream*, are provided by the Java framework. These classes provide no functionality of their own, and are intended to be subclassed to create new wrapper classes. In these cases, the constructor duplication and method forwarding can be avoided almost entirely.

2.3 Core Classes

The primary class in Furm is *ResourceNode*. The *ResourceNode* class encapsulates the complexities of adding resources to the tree, removing resources from the tree, releasing resources, synchronization, etc. The *ResourceGroup* class is a subclass of *ResourceNode* used for representing logical dependencies, as discussed in Section 2, and also as the root of the tree. The root of a resource tree must always be a *ResourceGroup*.

During construction, a *ResourceNode* instance will automatically add itself to the specified resource tree, and upon release, will propagate the *release()* call to its children. When a *ResourceNode* is released, it automatically removes itself from its parent. This allows garbage collection to proceed.

Starting from the *ResourceNode* on which the original *release()* call is made, the *release()* call propagates down the tree in post-order fashion. The *cleanup()* calls are executed while traversing back up to the root. The *ResourceNode* which has its *release()* method called first is thus the last to have its *cleanup()* method executed.

The fact that child resources are automatically released raises the question of how to deal with potential exceptions during this process. Java's ubiquitous exceptions make resource management more difficult, and a primary goal of Furm is to minimize this complexity. Child resources can potentially throw exceptions during release, but guaranteed cleanup is a necessity for Furm to properly replace the alternative – nested *try-finally* blocks – for releasing resources.

Furm gives the programmer a number of exception-handling choices. The first choice is which exceptions should be handled. Furm provides three *ListenLevels*: *CHECKED*, *EXCEPTION*, and *THROWABLE*. If *CHECKED* is specified, then only checked exceptions will be handled. Any other exceptions (instances of *Error* and *RuntimeException*) will go unhandled. This is similar to specifying *IOException* or *SQLException* in a *try-catch* block, but less fine-grained. If the specified *ListenLevel* is *EXCEPTION*, then all subclasses of *Exception*, both checked and unchecked, will be handled. Only *Error*s will go uncaught. If the specified *ListenLevel* is *THROWABLE*, then all subclasses of *Throwable* will be handled, i.e., both *Exception* and *Error*. The programmer has control over what level of exceptions to catch. The framework does not force any particular level. If the programmer chooses not to specify a *ListenLevel*, the default is *CHECKED*.

The second choice is what to do with the exceptions that need handling. For this purpose, Furm provides the *ExceptionListener* interface. Whenever a *release()* call is made on a *ResourceNode*, any exceptions which match the *ListenLevel* will be passed to an *ExceptionListener*. Any of the *ExceptionListener*s provided by Furm can be used, and new *ExceptionListener*s can be created by subclassing the *ExceptionListener* interface.

ExceptionListener is an interface with only one method, *exceptionThrown()*. Whenever an exception is thrown that meets the specified *ListenLevel*, it is passed to this method. The *ExceptionListener* interface can be described with very few lines of code, as shown here:

```
public interface ExceptionListener {
  public void exceptionThrown(ResourceNode n,
                              Throwable thr);
}
```

Creating a new *ExceptionListener* requires only one method to be overridden. For example, suppose that when a *SecurityException* occurs, we wish to log the system properties (Java version and vendor, classpath, etc.) for debugging purposes. Creating an *ExceptionListener* to do that requires only that a new subclass of *ExceptionListener* be defined with the *exceptionThrown()* method overridden. This new *PropertiesLogger* can be reused for multiple *ResourceNodes*, multiple resource trees, and multiple applications.

Three general-purpose *ExceptionListener* subclasses are provided by Furm: *ExceptionLogger*, which logs exceptions to a specified *PrintStream* or *PrintWriter*; *ExceptionCollector*, which builds a list of exceptions; and *ExceptionPasser*, which maintains a list of other *ExceptionListener*s and simply passes exceptions on to them. *ExceptionPasser* allows composition of *ExceptionListener*s.

Figure 2-10 illustrates the flow through a resource tree when cleanup code throws exceptions. This example assumes the default *ListenLevel* of *CHECKED*. Starting with a *release()* call made to the root of the tree, N_1, the *release()* call propagates down the tree. After a node's children have been released, its resource-specific *cleanup()* method executes. Note that nodes N_4 and N_5 both throw exceptions during cleanup. The *IOException* thrown by N_4 is a checked exception, and so is passed to the *ExceptionListener* for handling. The *NullPointerException* thrown by N_5 is an unchecked exception, and so is not passed to the *ExceptionListener*. Instead it propagates up the tree to the original caller. This does not affect the correct release of other resources.

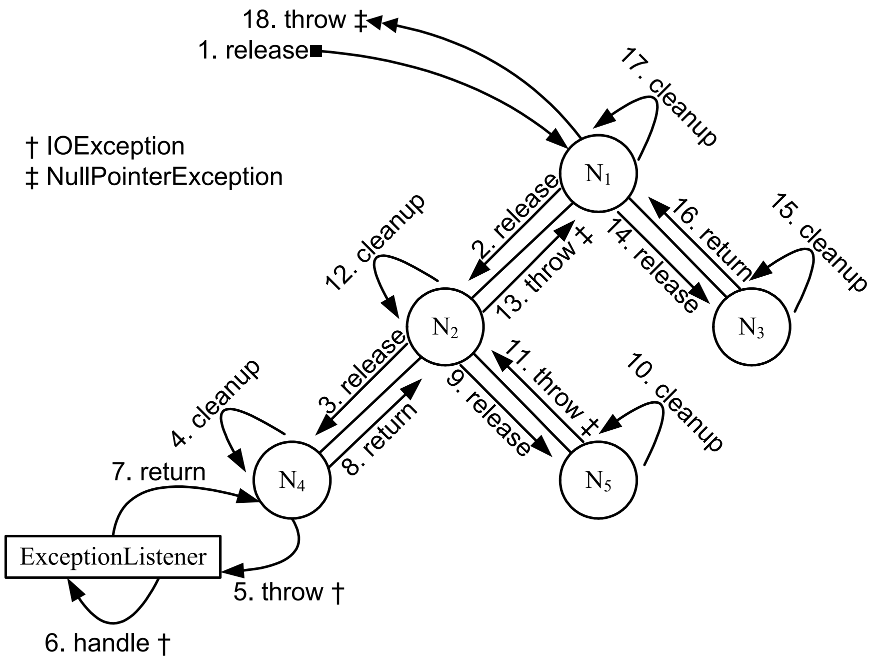


Figure 2-10. Propagation of *release()* calls with guaranteed cleanup

An *ExceptionListener* cannot rethrow any exceptions it receives, nor throw any exceptions of its own. Any exceptions thrown by *ExceptionListener*s are discarded.

Four *release()* methods in all are provided. The *ListenLevel* and *ExceptionListener* can be independently specified or left to defaults. This allows as little or as much control as a programmer needs.

public final int release()
public final int release(ListenLevel level)
public final int release(ExceptionListener listener)
public final int release(ExceptionListener listener, ListenLevel level)

The *release()* methods return the number of handled exceptions which occurred during release, including exceptions in child resources. This total includes only those exceptions which were passed to an *ExceptionListener*. Any exceptions that were not handled – those which did not match the *ListenLevel* – are not included in this total.

When a *ListenLevel* is provided as an argument to a *release()* call, it will be propagated down the tree with the *release()* call. If it is unspecified, the default will be used. Likewise, if an *ExceptionListener* is provided, it will be propagated down the tree. In the event that an *ExceptionListener* is not explicitly provided as an argument, the default *ExceptionListener* will be used. If the default has also not been set, then the ancestors of the *ResourceNode* will be searched for a default *ExceptionListener*, starting with the parent, and proceeding toward the root, where the default is guaranteed to have been set by the constructor. The closest *ExceptionListener* will be used.

2.4 Concurrency Concerns

The *ResourceNode* class is threadsafe. Any thread may create or release resources in any resource tree without the need for explicit synchronization. The *ResourceNode* class also provides a locking object for times when manual synchronization is necessary. If, for example, a need to explore the tree programmatically were to arise, it would be simple to safely navigate the tree in a *synchronized* block, as shown in Figure 2-11. The synchronization object is retrieved by calling *getMonitor()* on a *ResourceNode*. The monitor object is shared by the whole tree (but not among all trees), so synchronizing on the monitor object locks the entire tree. Such locking is safe, but inefficient. An alternative is proposed in future work.

This thread safety allows resource trees to be shared among threads. This has some resource profiling benefits. More importantly, this allows the resource tree metaphor to extend to complex, multi-threaded programs.

Suppose an application with multiple threads wishes to share a single tree among many threads. Whereas this would typically involve much manual synchronization, since concurrency is built into the *ResourceNode* class, this becomes straightforward. The shared tree can be partitioned into a number of logical subtrees. Each thread can have its own subtree, with an additional subtree shared by all threads. More complex schemes with subtrees shared by a subset of threads are also possible.

```
void lookAtChildren(ResourceNode parent) {
  synchronized (parent.getMonitor()) {
    if (parent.isReleased()) return;
    for (ResourceNode child : parent) {
      // use/investigate child node
    }
  }
}
```

Figure 2-11. Navigating a *ResourceNode*'s children

```
01 public void sendFile(ResourceNode parent,
02                      File f) {
03   try {
04     ManagedSocket s = new //...
05     FileInputStream in =
06         new ManagedFileInputStream(f,g);
07     ManagedOutputStream out =
08         s.getOutputStream();
09     byte[] buf = new byte[4096];
10     int len;
11     while ((len = in.read(buf)) >= 0) {
12       out.write(buf, 0, len);
13     }
14   } catch (IOException exc) {}
15   parent.release();
16 }
```

Figure 2-12. A "leaky" method

2.5 Automatic Resource Cleanup

When properly used, Furm opens up new possibilities for resource auditing. Furm provides a *WatchDog* class for monitoring threads for proper resource cleanup. A *WatchDog* is given a thread to guard, and a reference to the root of the resource tree (or subtree) being used by that thread. Whenever that thread dies, the *WatchDog* can take some action on any resources abandoned by that thread. This can be used as a safeguard to prevent resource leaks at runtime. The *WatchDog* can also be used as a resource profiler. If resource leaks are suspected or known to exist, every thread in a program can be assigned a *WatchDog* for tracing purposes. As threads die, any resources they leave open can be inspected. Information about those resources can aid in tracking down the leak.

There are two ways to control what action will take place when a thread dies. The first is to provide a *ThreadListener*. When a *WatchDog* observes that its thread has died, it will call the *threadDied()* method on the *ThreadListener*, much the way a *ResourceNode* calls *exceptionThrown()* on an *ExceptionListener*. To specify a *ThreadListener*, simply pass it to the *WatchDog* constructor. The second way to control what action the *WatchDog* should take is to extend the *WatchDog* class and override its *threadDied()* method. The *WatchDog* class itself implements the *ThreadListener* interface, and so has the *threadDied()* method.[5]

WatchDog has two *WatchLevel*s. The *WatchLevel* determines what forms of thread termination will be monitored. The two *WatchLevel*s are *exceptional* (declared as *EXCEPTIONAL_END*) and *general* (declared as *ANY_END*). If the *WatchLevel* is *exceptional*, then the *threadDied()* method will only be called if the thread dies by throwing an uncaught exception. If the *WatchLevel* is *general*, then the *threadDied()* method will be called regardless of the circumstances of thread termination.

As an example of how to use the *WatchDog* class, we present a small, "leaky" method which uses Furm. This method simply creates a socket connection and sends a file to the output stream of the socket. Figure 2-12 contains its full implementation.

[5] This model follows that of the *Thread* class, which has a constructor that accepts a *Runnable* object, but also implements the *Runnable* interface.

```
01 class Examiner implements ThreadListener {
02   private void printRes(ResourceNode res) {
03     Class c = res.getResource().getClass()
04     System.out.println(c.getCanonicalName());
05     for (ResourceNode rn : res) {
06       printResource(rn);
07     }
08   }
09   public void threadDied(Thread t,
10                          ResourceNode node,
11                          Throwable e) {
12     synchronized(node.getMonitor()) {
13       if (!node.isReleased()) {
14         printRes(node);
15         node.release();
16       }
17     }
18   }
19 }
20 // ...
21 new WatchDog(Thread.currentThread(),group,
22          new Examiner(),WatchLevel.ANY_END);
```

Figure 2-13. Utilizing *WatchDog*

Notice that the file transfer code is wrapped in a *try-catch* block which catches *IOException* (lines 3-14). The *release()* call is made on line 15. While this code will work in cases when no errors occur or when an *IOException* is thrown, if the *File* reference passed into the method is *null*, a *NullPointerException* will occur on line 6, and the *release()* call will never be made. The *Socket* and socket output stream will be leaked. The correct solution is to add a *finally* handler, but here this has not been done. If the code were part of a very large project, it may be reasonable to assume that no one has discovered the bug.

Let us assume that the *WatchDog* class is being used to hunt for a suspected bug, or that the *WatchDog* is simply being used as an additional safeguard in development code. To this end, the *Examiner* class (Figure 2-13) is added. Lines 21 and 22 create a *WatchDog* to monitor the thread. A *WatchDog* can be created to watch any thread. In this case, it is created to monitor the current thread. The *Examiner* class used here implements *ThreadListener* and performs two tasks in the *threadDied()* callback. First, it traverses the tree to discover any unreleased resources and prints that information. Second, it releases the tree.

An *Examiner* object is passed to the *WatchDog* constructor, and the *general WatchLevel* is specified. When the current thread dies, whether normally or as the result of an uncaught exception, the *Examiner* object will inspect the thread's resource tree. This simple auditing may reveal the problem with the *sendFile()* method. The *WatchDog* class is not a replacement for correct resource management, but it can make it simpler to track down cases of incorrect behavior.

One major benefit of the *WatchDog* class is that its use does not require watched threads to be aware of its presence. In fact, it is entirely possible for multiple *WatchDog*s to watch the same thread, each oblivious to the others, and the thread itself blind to them all. The main restriction on the use of the *WatchDog* class is that it requires that any thread monitored by a *WatchDog* place its resources in a resource tree. The *WatchDog* class cannot monitor resources which are not in a resource tree. Watched threads can ignore the *WatchDog*s, but must utilize Furm resource trees.

3. FUTURE WORK

The ability to split and merge resource trees is currently lacking in Furm. It might be useful to be able to split a resource tree when spawning a worker thread. Likewise, it could be useful to be able to merge resource trees when a thread finishes. Splitting and merging resource trees, if done correctly, could be an effective way to transfer resource management responsibility around in a program. Tree merging, in combination with a *WatchDog*, would allow a terminated thread's resource tree to be merged into the tree of a still-live thread. This would allow one thread to take responsibility for another thread's resources without the need to maintain two separate trees, as is currently required.

An additional area for future work involves optimization of the tree locking protocol. The current locking scheme simply locks the entire tree. This level of locking is conservative for guaranteeing safe concurrent access. More efficient and finer-grained locking protocols are known [11] which may be adaptable to Furm.

4. CONCLUSION

Because garbage collection and finalization are poorly suited to the task of releasing resources, the task falls on programmers to manually manage resource releases. This, combined with the complexities of exception-handling in Java, causes significant difficulty in the production and maintenance of correct applications. Any resource not guarded by a *try-finally* block can potentially be leaked if an unchecked exception occurs. In cases involving multiple resources, these *try-finally* blocks must be nested, with one level of nesting introduced for each resource. Complicating things further is that the Java API does not utilize a consistent interface for releasing resources, instead leaving each resource class to define its own methods for release.

The resource tree model introduced by Furm eliminates the inconsistent interfaces presented by Java. All resources are treated identically in the context of the tree, and so the method for releasing resources is consistent. Resource trees also represent resource dependencies, in which one resource is dependent on another. It is possible to safely prune entire trees with a single explicit release. It has been shown that Furm makes it possible to safely and correctly propagate release calls downward through a resource tree, while providing guarantees as strong as those of nested *try-finally* blocks.

The code overhead of explicit resource management in Java is high. When measured in lines-of-code using typical Java coding conventions, that overhead has been shown to be five lines of exception-handling to each line of resource cleanup. In programs utilizing numerous resources, that overhead can easily dominate the code, obscuring the primary purpose and logic of the application. Utilizing Furm can minimize this complexity. A single call can release an entire resource tree. By moving exception-handling logic into dedicated *ExceptionListener* objects, Furm also separates exception-handling code from the normal flow of code, making good on one of exception-handling's original promises.

5. REFERENCES

[1] Alexandrescu, A., and Marginean, P. Simplify Your Exception-Safe Code – Forever. *C/C++ Users Journal*, October 2000. http://www.ddj.com/dept/cpp/184403758 (accessed April 2006).

[2] Apache Tomcat Project. http://tomcat.apache.org/ (accessed April 2006).

[3] Bloch, J. *Effective Java*. Addison-Wesley, 2001.

[4] Campione, M., Walrath, K., and Huml, A. *The Java Tutorial.* 2005. http://java.sun.com/docs/books/tutorial/networking/index.html (accessed April 2006).

[5] Chen, R. Cleaner, more elegant, and harder to recognize. Jan. 14, 2005. http://blogs.msdn.com/oldnewthing/archive/2005/01/14/352949.aspx (accessed April 2006).

[6] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. A resource management interface for the Java platform. *Software: Practice and Experience*, 2005, pp 123-157.

[7] ECMA. *Standard ECMA-334: C# Language Specification*, 3rd Ed. December 2002. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf (accessed April 2006).

[8] Gosling, J., Joy, B., Steele G., and Bracha, G. *The Java Language Specification, 3rd Ed.* Addison-Wesley, 2005.

[9] Halloway, S. JDC Tech Tips: January 24, 2000. http://java.sun.com/developer/TechTips/2000/tt0124.html (accessed April 2006).

[10] Java Bug: 4484411, stack overflow error closing a socket input stream. 2001. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4484411 (accessed April 2006).

[11] Lanin, V. and Shasha, D. Tree Locking on Changing Trees. Technical Report 503, 1990, New York University.

[12] Park, D. A. Simplifying Resource Management in Java. Technical Report 2006-05. Master's Thesis, University of Mississippi, May 2006.

[13] Spolsky, J. How Microsoft Lost the API War. June 13, 2004. http://www.joelonsoftware.com/articles/APIWar.html (accessed April 2006).

[14] Spolsky, J. Making Wrong Code Look Wrong. May 11, 2005. http://www.joelonsoftware.com/articles/Wrong.html (accessed April 2006).

[15] Stroustrup, B. *The C++ Programming Language, 3rd Ed*. Addison-Wesley, 1997.

[16] Sun Microsystems. *Java 2 Platform Standard Edition 5.0 API Specification*. 2004. http://java.sun.com/j2se/1.5.0/docs/api/index.html (accessed April 2006).

[17] Weimer, W. and Necula, G. Finding and Preventing Run-Time Error Handling Mistakes. *Proc. of 19th ACM OOPSLA*, pp 419-431, Vancouver, British Columbia, Canada, October 2004.

Session E
Software Engineering

Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development

Matthias Merdes
EML Research gGmbH
Villa Bosch
Schloss-Wolfsbrunnenweg 33
D-69118 Heidelberg, Germany
<firstname.lastname>@eml-r.villa-bosch.de

Dirk Dorsch
EML Research gGmbH
Villa Bosch
Schloss-Wolfsbrunnenweg 33
D-69118 Heidelberg, Germany
<firstname.lastname>@eml-r.villa-bosch.de

ABSTRACT

The development of a tool for reconstructing UML sequence diagrams from executing Java programs is a challenging task. We implemented such a tool designed to analyze any kind of Java program. Its implementation relies heavily on several advanced features of the Java platform. Although there are a number of research projects in this area usually little information on implementation-related questions or the rationale behind implementation decisions is provided. In this paper we present a thorough study of technological options for the relevant concerns in such a system. The various options are explained and the trade-offs involved are analyzed. We focus on practical aspects of data collection, data representation and meta-model, visualization, editing, and export concerns. Apart from analyzing the available options, we report our own experience in developing a prototype of such a tool in this study. It is of special interest to investigate systematically in what ways the Java platform facilitates (or hinders) the construction of the described reverse engineering tool.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *object-oriented design methods*, D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *reverse engineering, documentation.*

General Terms

Algorithms, Documentation, Design, Experimentation

Keywords

UML models, sequence diagrams, reverse engineering, Java technology

1. INTRODUCTION

Due to the increasing size and complexity of software applications the understanding of their structure and behavior has become more and more important. Proper specification and design activities are known to be important in producing understandable software. If such specification and design artifacts are unavailable or of poor quality reverse engineering technologies can significantly improve understanding of the design of an existing deployed software system and in general support debugging and maintenance. While modern CASE tools usually support the reconstruction of static structures, the reverse engineering of dynamic behavior is still a topic of on-going research [20], [25].

The development of a tool supporting the reconstruction of the behavior of a running software system must address the major areas of data collection from a (running) system, representation of this data in a suitable meta-model, export of the meta-model's information or its graphical representation as well as post-processing and visualization aspects. These core areas and their mutual dependencies are shown in Figure 1. Clearly, all conceptual components depend on the meta-model. In addition, a visualization mechanism can be based on a suitable export format as discussed in sections 4 and 5. While this figure illustrates the main conceptual components of our sequence diagram reengineering tool a symbolic view of its primary use can be seen in Figure 2: The main purpose of such a tool is to provide a mapping from a Java program to a UML sequence diagram. The various relevant options will be discussed in detail in the following sections. Recurrent technical topics include meta-model engineering, aspect-oriented technologies, XML technologies – especially in the areas of serialization and transformation – and vector graphics.

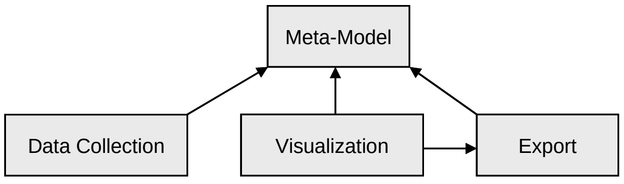
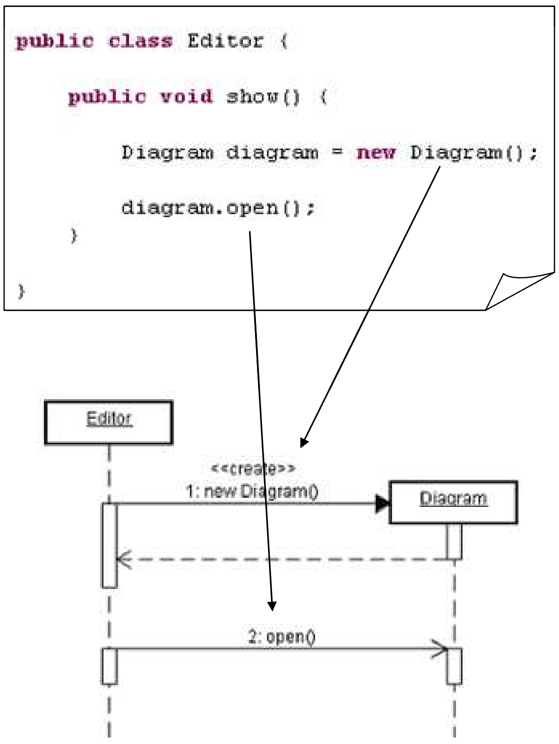


Figure 1. Conceptual components with dependencies

UML sequence diagrams are among the most widely used diagrams of the Unified Model Language (UML) [32]. The UML is now considered the lingua franca of software modeling supporting both structural (static) and behavioral (dynamic) models and their representation as diagrams. Behavioral diagrams include activity, communication, and sequence diagrams. Such sequence diagrams are a popular form to illustrate participants of an interaction and the messages between these participants. They are widely used in specification documents and testing activities [24] as well as in the scientific and technical literature on software engineering.

Sequence diagrams [32] are composed of a few basic and a number of more advanced elements. The basic ingredients of a sequence diagram are illustrated in a very simple example in the right part of Figure 2 along with their respective counterparts in the Java source code on the left-hand side. In such a diagram participants are shown along the horizontal dimension of the diagram as so-called ‘life-lines’. In the example, the two participants are ‘Editor’ and ‘Diagram’. These life-lines are connected by arrows symbolizing the messages exchanged between participants. The messages are ordered chronologically along the vertical dimension. In the example, two messages from Editor to Diagram are depicted, namely the constructor message ‘new Diagram()’ and the ‘open()’ message. More advanced concepts (not shown in the figure) such as modeling alternatives, loops, and concurrent behavior, can be factored out into so-called ‘fragments’ for modularization and better readability.



```
public class Editor {

    public void show() {

        Diagram diagram = new Diagram();

        diagram.open();
    }

}
```

Figure 2. Behavior as Java source code and sequence diagram

The reconstruction of the behavior of a software system has been studied extensively both in the static case (from source or byte code) [36], [37], [38] and in the dynamic case (from tracing running systems) [6], [33], [34]. [42] and [7] focus more on interaction with and understanding of sequence diagrams, respectively. An overview of approaches is provided by [25] and [20]. Despite this considerable amount of work there is often little information on implementation-centric questions or the rationale behind implementation decisions. Our study is intended to remedy this lack of such a systematic investigation and is motivated by our experiences in implementing our own sequence diagram reengineering tool. This paper has two main purposes. Firstly, we describe and analyze the possible technological options for the required areas. We also report the lessons learned by our implementation. In this way, the more abstract analysis based on theoretical considerations and the technical and scientific literature is verified and complemented by our own practical experience.

The remainder of this paper is organized as follows. Section 2 explores methods to collect relevant data and section 3 describes the choices for representation of this data using a suitable meta-model. We describe options for visualization and model or graphics export in section 4 and 5, respectively.

2. Data Collection

In this section we will discuss technologies for retrieving information from Java software systems with the purpose of generating instances of a meta-model for UML sequence diagrams. We focus on dynamic (or execution-time) methods but cover static (or development-time) methods as well for the sake of completeness. Static methods gather information from a non-running, (source or byte) code-represented system. Dynamic methods on the other hand record the interaction by observing a system in execution. Data collection requires a mechanism for filtering relevant execution-time events which supports a fine-grained selection of method invocations.

2.1 Development-time Methods

2.1.1 Source Code Based

Using the source code for collecting information about the interaction within an application will have at least one disadvantage: one must have access to the source code. Nevertheless source code analysis is a common practice in the reverse engineering of software systems and supported by most of the available modeling tools. It should be mentioned that the analysis of source code will provide satisfactory results for static diagrams (e.g., class diagrams), but the suitability for the dynamic behavior of an application is limited. If one is interested in a sequence diagram in the form of a common forward engineered diagram (i.e., a visualization of all possible branches of the control flow in the so-called CombinedFragment [32] of the UML), source code analysis will fulfill this requirement. In [37] Rountev, Volgin, and Reddoch introduce an algorithm which maps the control flow to these CombinedFragments. If the intention of the reverse engineering is to visualize the actual interaction any approach of static code analysis is doomed to fail, since it is inherently not possible to completely deduce the state of a system in execution by examining the source code only without actually running the system. Obvious problems include conditional behavior, late binding, and sensor or interactive user input.

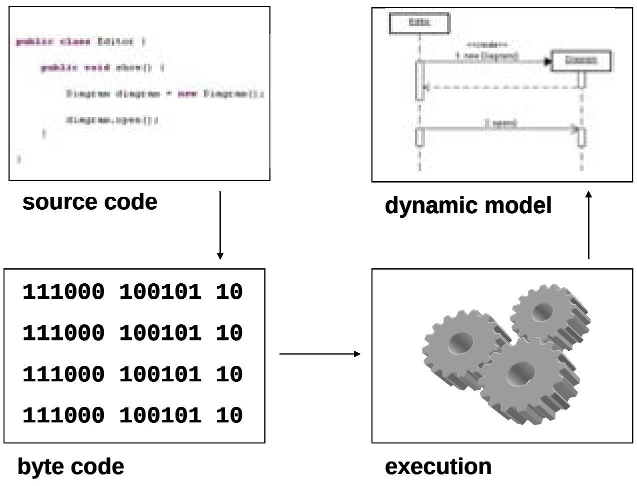
2.1.2 Byte Code Based

The static analysis of code can also be performed with compiled code, i.e., byte code in the case of Java. Such an analysis of byte code basically shares most of the (dis-) advantages of the source code based approach, but it can be applied to compiled systems. One advantage is the fact that processing the byte code must be performed after compilation, separate from the source code, and thus leaves the source code unchanged. This prevents mixing of concerns (application logic and tracing concerns) in the source code and connected maintenance problems.

2.2 Execution-time Methods

The purpose of the dynamic approaches is to record the *effective* flow of control, or more precisely, the sequence of interactions, of a (deployed) system's execution. Any dynamic approach results in a model that represents the actual branches of the application's control flow. In this section we will discuss technologies based on a temporary interception of the program's execution. Basically, we differentiate between the instrumentation of the application itself (i.e., its code) and the instrumentation of its runtime environment.

An overview of the basic workflow from the Java sources to the byte code and on to the UML model and its visualization can be seen in Figure 3. This figure illustrates the more expressive approach of generating the model from dynamic runtime trace information, compared to the static approach described in section 2.1, which relies on source code only.



```
public class Editor {
    public void show() {
        Diagram diagram = new Diagram();
        diagram.open();
    }
}
```

```
111000 100101 10
111000 100101 10
111000 100101 10
111000 100101 10
```

Figure 3. Symbolic steps from source code to sequence diagram model for a Java program (dynamic analysis)

2.2.1 Program Instrumentation

2.2.1.1 Source Code Based

Assuming access to the source code is provided it can be instrumented in a number of ways. Two obvious possibilities are:

1. Modify the source code manually; this is both troublesome and error-prone.

2. Take advantage of aspect-orientation and compile the code with suitable aspects.

Both will finally result in modified source code either explicitly or transparently. Support for filtering can be achieved by a (manual or automatic) manipulation of selected source code fragments. Another related approach is the common logging practice which can be seen as source code manipulation as well. Such an analysis of log-files is discussed in [17].

2.2.1.2 Byte Code Based

Instrumenting the byte code instead of the source code has one advantage: the source code is not manipulated in any way. Again, one could take advantage of aspect-orientation and recompile the byte code with some aspects [5]. In most cases one will have access to the byte code in the form of Java archives (jar files) or raw class files; otherwise this approach will fail. Again, as in the development time case explained in section 2.1.2, byte code manipulation is superior to source code manipulation because of maintenance and versioning issues. In the following section another aspect-oriented approach will be discussed.

2.2.2 Instrumentation of the Runtime Environment

For Java applications the instrumentation of the runtime environment means the instrumentation of the Java Virtual Machine (JVM). When discussing JVM instrumentation the theoretical possibility to develop a customized JVM should be mentioned. Due to the large effort of implementing a new or even modifying an existing virtual machine we won't discuss this approach any further. We prefer to introduce technologies based on virtual machine agents that could be applied to existing JVM implementations. In principle, a custom agent could be developed against the new Java Virtual Machine Tool Interface (JVMTI), which is part of J2SE 5.0. Gadget [16] is an example using an older version of this API for the purpose of extracting the dynamic structure of Java applications. Using the AspectJ or Java-Debug-Interface (JDI) agents as described below allows to focus on a higher level of abstraction compared to the low-level tool interface programming.

2.2.2.1 Java Debug Interface (JDI)

The JDI is part of the Java Platform Debugger Architecture (JPDA) [45]. The JPDA defines three interfaces, namely the Java Virtual Machine Tool Interface (JVMTI, formerly the Java Virtual Machine Debug Interface, JVMDI) which defines the services a virtual machine must provide for debugging purpose, the Java Debug Wire Protocol (JDWP) which defines a protocol allowing the use of different VM implementations and platforms as well as remote debugging, and last but not least the JDI, the Java interface implementation for accessing the JVMTI over JDWP. The debuggee (in our case the observed program) is launched with the JDWP agent, this allows the debugger (in our case the observing application) to receive events from the debuggee by using JDI. For the purpose of reengineering the system's behavior we are mainly interested in events of method executions. As shown in JAVAVIS [33] the JPDA could be successfully used for the purpose of dynamic reverse engineering. One big advantage of the JPDA is the built-in remote accessibility of the observed application. The event registration facility, which can be seen as a filtering mechanism, appears to be too coarse grained, since the class filter is the finest level of granularity. Nevertheless, the JPDA permits the development of reverse engineering tools for both, structural (static) models, such as class diagrams, and behavioral (dynamic) models, such as sequence diagrams.

2.2.2.2 *AspectJ Load Time Weaving*

Usually aspect-oriented programming is associated with recompiling the source code or byte code with aspects (a.k.a. weaving), as mentioned in section 2.2.1. Starting with version 1.1, the AspectJ technology also offers the possibility of load-time-weaving (LTW) where the defined aspects are woven into the byte code at the time they are *loaded* by the class loader of the Java virtual machine [12]. Hence AspectJ offers the possibility to trace a deployed system without modifying either source code or byte code.

An extensive discussion on how to use AspectJ for the purpose of dynamic reverse engineering of system behavior can be found in [5] and is beyond the scope of this paper. In this section we therefore restrict ourselves to the discussion of the basic concepts of AspectJ needed for this purpose. For detailed information about aspect-orientation and especially AspectJ refer to [15], [23], and [12]. Recent research results and directions can be found in [13].

Generally, aspect-oriented approaches support the modularization of cross-cutting concerns with aspects and weaving specifications. In the case of AspectJ, these concepts are realized by aspects (comparable to classes), advice (comparable to methods) and joinpoints specified by pointcut descriptors. An advice declares what to do before (*before advice*), after (*after advice*) or instead of (*around advice*) an existing behavior addressed by a specific joinpoint. The joinpoint exactly defines a point within the code execution. For retrieving the information needed to model a sequence diagram it is sufficient to take advantage of the predefined *call joinpoints* (representing a method call) and *execution joinpoints* (representing a method execution).

The definition of a joinpoint also offers the possibility of filtering. A joinpoint can address packages, classes, selected methods or work in an even more fine-grained manner. So combining those joinpoints and the arbitrary granularity of the filter mechanism allows for a flexible extraction of the information on the interactions in a running system.

2.3 Comparative Assessment

As presented in the preceding sections, there are numerous ways to implement an execution-tracing data collection mechanism. Discriminating dimensions include manual vs. automatic instrumentation of source or byte code, static vs. dynamic analysis, remote accessibility and performance issues.

If the target environment allows the combined use of version 5 of the Java platform and the latest release of the AspectJ distribution (AspectJ 5) the elegance and non-intrusiveness of the load-time-weaving mechanism in combination with the low performance impact and the expressiveness and flexibility of the join-point-based filter mechanism make the aspect-oriented approach the best solution. This approach is superior in all relevant dimensions, especially compared to the manual integration of tracing and application code due to associated maintenance problems, and compared to a custom JVM or custom JVM agents due to their inherent complexity and huge effort. Hence in our tool we use an AspectJ-based data collection mechanism but we have also implemented and evaluated a prototypical JDI-based data collection mechanism. Such a solution, however, requires a customized filtering extension to achieve an appropriate filtering granularity and suffers from performance problems, especially in the presence of graphical user interfaces.

3. Meta-Model and Data Representation

A central topic which influences other areas, e.g., visualization, editing, or export, is the question of how the recorded data are represented internally. This is best achieved by storing the data in instances of a suitable meta-model. As a sequence diagram generation tool collects information on the *execution* of a program the meta-model must be capable of representing such run-time trace data.

Of course, only a certain subset of a typical complete meta-model will be needed for representing the relevant data. As the execution of a program in an object-oriented language is realized by method calls between sender and receiver with arguments, return types, and possibly exceptions, these are the required meta-model elements. Therefore a compatible meta-model must be employed rather than the actual meta-model of the programming language. Specifically, for an object-oriented programming language like Java a generalized object-oriented meta-model can be used, such as the OMG meta-model, the Meta-Object Facility (MOF) [30], to which other languages than Java can be mapped as well. Meta-models are at the core of recent research and standardization activities in the area of the OMG's Model Driven Architecture (MDA) [28], [39] and, more generally, Model Driven Development (MDD) which encompasses approaches beyond the OMG standards, such as Domain Specific Languages (DSLs) [19] and other non UML-based efforts.

3.1 Meta-Models from MDD Tools

MDD technologies usually generate executable software from instances of meta-models [46]. That implies that tools for such technologies need a representation of the respective meta-model. An example is the free openArchitectureWare Framework (OAW) [48] which includes a meta-model implementation in Java. One of the advantages is that exporting the meta-model to various formats is supported including a number of XMI dialects. The decision for the use of such a meta-model is a trade-off between the advantages of reusing a stable quality implementation and the overhead involved with a much larger meta-model than needed and a certain amount of inflexibility due to the external dependency (e.g., reliance on third-party bug fixing or version schedules).

3.2 UML2 Meta-Model

Since the UML2 specification [32] defines 13 types of diagrams and a large number of classes it would be quite expensive to implement the full UML2 meta-model from scratch. The EclipseUML2 project [9] provides a complete implementation of the UML2 meta-model. It is based on the Eclipse Modeling Framework (EMF) and its meta-model Ecore [10]. While the EMF was designed as a modeling framework and code generation utility the objectives of EclipseUML2 are to provide an implementation of the UML meta-model to support the development of UML based modeling tools. EclipseUML2 and the EMF also provide support for the XML Metadata Interchange language (XMI) [31] export of the model. This built-in XMI serialization is a big advantage for the model exchange as described in section 5.1.1. Despite those advantages the usage of the UML2 meta-model for representing only sequence diagrams could cause some cognitive overhead as most parts of the UML structure won't be needed.

3.3 Custom Meta-Model

The overhead produced by using the complete UML2 meta-model leads to the idea of developing a light-weight custom meta-model. As mentioned in the introduction one can reduce the model to a few basic components which will result in a very light-weight design. However, one has to face the drawbacks of developing an export mechanism in addition to persistence and visualization mechanisms.

3.4 Comparative Assessment

As the central abstraction in a sequence diagram reverse engineering tool is the data about the recorded sequences from actual program runs, its representation in instances of a meta-model is a crucial question. There are two basic options to choose from: reusing an external meta-model or implementing a custom meta-model. The reuse of an external meta-model offers the well-known substantial advantages of software reuse [21], such as implementation cost savings and proven implementation quality. From a Java perspective both options are equally viable: Third-party meta-model implementations are very often based on Java technology and Java is also well suited for a custom implementation. In the given situation where only a very small subset of the meta-model is needed and the cost of a custom implementation is low we opted for the simplicity and flexibility of a custom implementation.

4. Visualization and Model Post-Processing

One central requirement for a UML2 sequence diagram reverse engineering tool is the visualization of the recorded data, that is, some form of transformation or mapping from the meta-model instances to a visual representation. Indeed, as a human observer interacts with such models primarily in their visual form the *graphical display* as a sequence diagram can be considered the main purpose of such a tool. In the following sections we will discuss the possibilities of generating diagrams after recording the tracing data and analyze a number of possible methods. We also describe interactive rendering during the data recording.

4.1 Third-Party Batch Visualization

Methods based on third-party tools visualize the collected model information by exporting to viewable formats or intermediate stages of such formats. Generally, we can differentiate between using common graphics formats (such as PNG, JPG etc.) and displaying the result in third party UML modeling tools.

The main drawback of using static graphics formats is the lack of adequate diagram interaction possibilities. As bitmap formats offer the most simple export of a visualized diagram we will briefly explain a lightweight technology for generating various kinds of graphics output. The free tool UMLGraph is an example of such a technology. It allows the specification of a sequence diagram in a declarative way [40], [41] using pic macros. With the GNU plotutils program pic2plot [18] these specifications can be transformed into various graphics formats (such as PNG, SVG, PostScript, and many more). An approach for integrating this technology into a tool is the usage of a template engine (e.g., Apache Jakarta Velocity [2]) for transforming the instances of the meta-model to the pic syntax and applying pic2plot to the result. Main advantages include the implicit use of a high-quality layout engine and the broad graphics format support. Generally, all methods described in section 5.2 that lead to graphics export can be used for such batch visualizations.

4.2 Real-Time Visualization

It is an interesting option to perform model visualization during the data collection process. Especially for slower running or GUI input driven programs this can be a useful way of observing the behavior of the program in real time during the recording process. In [33] a similar approach is taken and combined with explicit means to trigger the steps in the program execution resulting in a special kind of visual debugger.

A number of implementation choices exist, especially the development of a custom viewer and an SVG-based solution. Although SVG is better known as a vector format for static graphics it also supports the visualization of time-dependent data in the form of animations [50]. For this purpose it is possible to programmatically add nodes to the SVG tree to reflect the temporal evolution of the diagram. In principle, the same two possible SVG-based approaches as those detailed in section 4.3.1 can be used.

4.3 Interactive Visualization and Editing

Viewing a non-modifiable diagram can already be useful. For diagrams constructed manually with a design tool this may be sufficient because these diagrams are usually not overly large as the content is under direct and explicit control of the modeler. If, however, the diagram is generated automatically by collecting data from an executing program it can quickly become very large. This may be caused by too many participants (lifelines) in the diagram or by recording too many interactions over time or by showing too much or unwanted detail. As pointed out by Sharp and Rountev [42] such a large diagram quickly becomes useless to a human user and thus a possibility to interactively explore the diagram is needed. Such an interactive viewing can in principle be extended to support the editing and modification of a diagram for further export. We describe three possibilities for realizing an interactive interface in the following sections.

4.3.1 SVG Based Solution

A viewer for the interactive exploration and possibly manipulation of sequence diagrams can be realized with Scalable Vector Graphics (SVG) [50]. We describe this W3C standard-based vector graphics format in more detail in section 5.2.2. The two principle possibilities are:

1. SVG combined with EcmaScript
2. Extension of an open source SVG viewer

In the first case the model is exported to an SVG image and combined with an embedded EcmaScript program for interaction. The scripting language EcmaScript [8] is the standardized version of JavaScript. While the latter was originally introduced by Netscape for client-side manipulation of web pages and the browser, EcmaScript is a general-purpose scripting language. It is the official scripting language for SVG viewers with standardized language bindings [52]. EcmaScript provides mouse event handling and enables the manipulation of SVG elements, attributes, and text values through the SVG Document Object Model (DOM). Nodes can be added and deleted and values modified. As SVG elements can be grouped during the export process and attributes can be inherited it becomes feasible to manipulate a whole UML sequence diagram element with a single attribute change in the parent group. It is beneficial that such an EcmaScript-based interactive explorer can be embedded into (or referenced from) the SVG file. Thus the image can be explored interactively in every SVG-compatible viewer including web browsers equipped with an SVG plug-in. Disadvantages of such a scripting language compared to a high-level object-oriented programming language like Java include limitations of the core language libraries, as well as fewer third-party libraries and generally comparatively poor (though slowly improving) tool support for EcmaScript development.

As an alternative an interactive viewer can be based on a Java implementation of an SVG library, such as the Apache Batik Toolkit [3] which includes parser, viewer, and an implementation of the Java language bindings according to the standard [51]. This toolkit is an open-source product which can be extended to support custom behavior either by modifying the existing viewer or by adding event-handlers with DOM-manipulating custom behavior to the view leaving the core viewer unmodified. While the first possibility as described in the preceding section is more powerful it requires changes to the original code which is a potential source of maintenance problems. The second approach is comparable to the one described for EcmaScript but with the greater power of Java compared to EcmaScript. The required manipulation of the DOM is possible but sometimes troublesome. The main advantage of using the Java API of Batik is the possibility to reuse a stable production-quality and free implementation. This approach to extend the existing Batik SVG viewer with custom interaction possibilities is described in [29] for the display of interactive maps within the Geographic Information Science (GIS) domain.

4.3.2 Custom Viewer

The most flexible approach is to build a custom viewer from scratch in Java, or even based on diagramming libraries such as the Graphical Editing Framework (GEF) [11] or JGraph [22]. The advantage of this approach is that the structures in a sequence diagram can be manipulated at the appropriate level of abstraction. In the SVG implementation manipulation of sequence diagram elements requires manipulation of the geometric representation of these elements. In that case the programmatic interaction is at the wrong level of abstraction, namely at the level of the graphical presentation and not at the level of the model itself. With a custom viewer, however, the display can be modified as response to user input by manipulating instances of the meta-model and their graphical representation. This can be achieved by adhering to the well-known Model-View-Controller (MVC) paradigm [26], a combination of the Observer, Composite, and Strategy design patterns [14] which promotes better design and maintainability. In this design changes can be applied to the model and automatically reflected in the graphical representation.

The drawback to this approach is primarily the fact that the rendering has to be implemented from scratch using only the basic abstractions such as points, lines, and text provided by the programming language, in this case Java or a suitable library. For a more complex interactive viewer, which may support hiding, folding, deleting of structures, or zooming and other manipulations, the greater expressiveness and power of Java (compared to SVG-viewer embedded EcmaScript) clearly outweighs this disadvantage. Additionally, if the model storage and diagram visualization concerns are handled within the same technology the overhead for and complexity of interfacing between technology layers (e.g., between Java and SVG/EcmaScript) can be saved. This is especially important for advanced interaction features which require semantic information stored in the model.

4.4 Comparative Assessment

In this section we described various sequence diagram visualization options and technologies. These include batch, real-time, and interactive visualization. While the batch mode provides basic visualization support, the usefulness of a sequence diagram reengineering and visualization tool is greatly increased if real-time and interactive visualization are supported. Thus, our tool also supports these two advanced options. The described SVG-based approaches have mainly the following advantages:

- Reuse of the base rendering mechanism of commercial (SVG browser plugins) or open source viewers (e.g., Batik)
- Ubiquitous deployability in the case of an EcmaScript-based viewer embedded within the SVG document due to readily available web browser plugins

However, these advantages are reduced by the cost and effort of bridging the technology gap between the recording and model storage technology (Java) and the viewing/rendering technology (EcmaScript/SVG). Especially for the advanced interaction features of our tool the flexibility of a custom viewer is crucial. We therefore decided to implement a custom viewer based solely on Java without an SVG-based implementation.

5. Export

In a model reengineering tool the model information is represented at different levels including an abstract non-visual level for the core model information and a more concrete level for the visual representation in a graphical user interface. The information at both levels has a distinct value for its respective purpose and therefore a tool should be able to export this information at both levels. Additionally, a third possibility is to export an animated version of the model. Such an animation combines the graphical representation with a temporal dimension thus capturing some of the actual dynamics encountered during the recording phase.

5.1 Model Information Export

Models are exported for a number of reasons including:

1. Import into other UML tools

2. As source for transformations to other representations, such as content-only (e.g., graphics) or textual model descriptions (e.g., DSLs)

3. As a persistence mechanism for the modeling application if it allows some form of editing or manipulation the state of which might need to be persisted

Options for such an export are XMI export, JavaBeans XML export, or custom (possibly binary) file formats. We describe each option briefly in the following.

5.1.1 XML Metadata Interchange (XMI)

UML models can be represented and shared between modeling tools and repositories with the help of the XMI standard defined by the OMG [31]. This standard is quite comprehensive and has evolved considerably to the current version. However, the existence of various dialects of the standards (as evidenced by the different model import filters of some modeling tools), constitutes a major problem for interoperability.

5.1.2 XML Data-Binding Based Serialization

An alternative export of model information can be accomplished by using the default XML serialization mechanism of the Java language. The initial Java Beans serialization mechanism was a binary one (see next section), which was and still is useful as a short-time serialization mechanism, e.g., for RMI remoting. It is very sensitive to versioning problems and unsuited to processing by non-Java technologies. Due to these problems and to the general growing importance of XML technologies, and in order to support long-term persistence a new XML-based serialization mechanism for Java Beans was added to the language in version 1.4 [44].

In light of the XMI interoperability problems the robustness, availability, and simplicity of this serialization mechanism can outweigh its limitations, namely the missing import capabilities into third-party modeling tools, especially in connection with a custom meta-model. The advantages of this serialization mechanism are limited to certain situations where (light-weight) models are created for documentation or ad-hoc communication purposes. This mechanism should not be used for creating persistent software lifecycle artifacts where model interchange is crucial.

5.1.3 Custom File Format

A binary custom (with respect to the contents not to the general structure) file format can be realized easily. To this end, the mentioned *binary* Java serialization mechanism is applied to modeling information represented in memory as an instance of the meta-model. The usefulness of such an export is quite limited, however, and can mainly be used as a proprietary persistence format for the application itself. It is not well suited for further processing or exchange with other tools, mainly due to its non-self-describing syntactic nature (i.e., binary Java persistence format) and missing meta-model (i.e., the static design of the stored objects).

5.2 Graphics Export

For many users and usage scenarios the export of modeling information is not needed; the export of images is sufficient. As mentioned earlier sequence diagrams play an important role in software specification and documentation artifacts as well as a basis for test coverage criteria [4], [24]. For the use within these documents and activities a visual form is needed and therefore a possibility to export diagram representations of the model both as static graphics and as animated diagrams.

5.2.1 Bitmaps

Bitmaps are the most simple of graphics formats and a large number of formats exist. The most popular formats include uncompressed bitmaps like Windows bitmaps (BMP) or TIFF bitmaps and compressed (lossy or lossless) formats like GIF, PNG, and JPEG. The main advantages of these formats include their wide-spread use, graphics tool and native browser support. The most popular formats like GIF and JPEG are also directly supported in programming languages like Java without third-party libraries or filters. Due to the discrete nature of the information encoding the contents of such an image cannot in general be scaled (or rotated by arbitrary angles) without lowering the image quality. This is especially true for drawings and text which are the constituents of sequence diagrams. Thus, bitmaps are primarily useful for direct processing between applications (e.g., via screenshots and clipboards), general screen-based use or medium-quality printed documentation but not necessarily for high-quality printing, such as books etc.

5.2.2 Vector Graphics

Vector graphics do not suffer from the inherent limitations of bitmaps with respect to image manipulations such as zooming. The structures in vector graphics images are not rastered but represented by a more abstract description on the level of lines for general drawings and letters for text. This enables reproduction at arbitrary resolutions and in many cases also leads to a smaller file size. Vector graphics formats exist in proprietary versions such as Adobe’s PostScript (PS), Encapsulated PostScript (EPS), and PDF formats or open-standards based versions, notably the W3C’s Scalable Vector Graphics (SVG) [50]. They also can be differentiated by their binary (PDF) or textual representation (SVG, PS). Of these formats SVG is the only XML-based format.

Although the Adobe family of formats is proprietary it is very widely used for electronic documents (PDF) [1] with the free Adobe Acrobat viewer, printers (PS), and within the printing industry. So sequence diagrams exported to PDF are immediately useful for sharing, viewing, and printing. Although free [27] as well as commercial programming libraries [35] for the generation of these formats exist the known disadvantages (e.g., legal as well as technical issues) of a proprietary format are most relevant for the creation process. Also the level of abstraction in these libraries varies and the API itself is not standardized. In principle, PDF can be generated directly at a high level of abstraction with the help of XSL formatting objects (XSL-FO) [49]. These formatting objects can be applied to a serialized form of an instance of the meta-model. Interestingly, this part of the XSL specification has enjoyed far less success than the XSL transformation part and is not widely supported. However, there is a fairly advanced implementation called FOP (Formatting Objects Processor) within

Apache's XML Graphics Project. A custom implementation of, e.g., a PostScript export is not advisable as the complexity and investment can be considerable.

The SVG standard [50] is a fairly recent development by the W3C consortium. The current production version 1.1 includes rich support for static vector graphics including text and bitmap integration as well as support for animation. As an XML-based format it is widely supported in the areas of generation (data-binding), parsing and transformation technologies (XSLT, XSL-FO) and provides very good editor, modeling tool, and persistence support. While these are generic advantages of XML-based formats special support for SVG is also growing in the area of viewers (e.g., Apache Batik) and browser plug-ins (e.g., Adobe, Corel) and as persistence format in many graphics tools. The Apache Batik Project [3] also provides a Java API for SVG parsing, generation, conversion, and a DOM implementation compliant to the standard.

These properties make SVG a suitable format for the export of models as diagrams. Additionally, SVG supports a generic extension mechanism for handling metadata [50]. In principle, this could be used to embed model information – possibly directly in XMI format – or processing status and history into the diagram representation. The SVG file could then be used as both an image format and a persistence format for the modeling application itself. An example of embedding domain knowledge at the level of model information into SVG metadata is described in [29] for the geographic domain. Additionally, SVG supports vector graphics based animation, which we describe in the next section.

5.3 Animation

The usefulness of animation as a tool for improving the understanding of sequence diagrams has been studied by Burd et al. [7], who find that control flow comprehensibility can be improved by using animated sequence diagrams compared to static sequence diagrams. There also seems to be initial support for such animated diagrams in commercial products [47].

The consideration between animated bitmaps (GIF) and vector graphics (SVG) is similar to that for the case of static diagrams. While the support of animated GIFs in browsers and, generally, in tools is better, SVG animation offers the known advantages of vector graphics, i.e., smaller file size, better quality, and scalability for text and drawings. Additionally, SVG animations are just XML files and could thus be easily post-processed by, e.g., XSL transformations to generate different representations, such as textual descriptions of the sequence or series of single images.

5.4 Comparative Assessment

For a sequence diagram reengineering tool export possibilities are very important. This includes export of both the semantic (model) information as well as a visual description (image data). Despite the well-known practical XMI interoperability problems support for this model exchange format is mandatory for a modeling tool. The built-in XML and binary serialization formats of the Java language provide useful mechanisms for intermediate storage (e.g., for model transformations) and for proprietary persistence formats, respectively.

Graphics export support includes more widely-used bitmaps, such as GIF with associated scaling and printing problems, and the less common but more scalable vector graphics formats, such as SVG. With this mix of advantages and drawbacks there is no single solution but support of both kinds of formats is useful. Animation export possibilities are a useful enhancement which can contribute to the improvement of model and, hence, program logic comprehension.

To support model information persistence for the application itself, we opted to use the JavaBeans built-in XML format. This also offers the possibility to easily extend the export to XML-based standards, like XMI and SVG, by applying XSL Transformations [53] to the serialized model. We will use this approach to support at least one important XMI dialect as part of our future work.

As with the other concerns, the mixture of features built into the Java language and the availability of third-party libraries and interfaces provide a strong foundation for the tool implementation.

6. Conclusion and Future Work

In this paper we presented a detailed study of technological choices for various implementation aspects of a dynamic UML sequence diagram reengineering tool from a Java-centric perspective. The implementation of such a tool presents a considerable challenge and many important strategic and technological decisions have to be made. Our study complements the existing body of literature on the subject of sequence diagram reengineering, or more generally, trace visualization, by adding thorough technical advice for those interested in attempting the implementation of such a system, especially in the Java language. In many cases there is no one single correct technological solution to a given implementation problem. By comparing the advantages and drawbacks of each alternative and reporting experiences from our own implementation this study provides assistance for informed technological decisions within the domain of Java-based sequence diagram reengineering, especially in the areas of data collection, data representation with meta-model instances, interactive model visualization and various export options.

We showed that Java is a very suitable language for the development of such a tool in two respects: While the virtual machine-based execution mechanism provides excellent support for tracing mechanisms for data collection, the many advanced features of Java discussed above as well as the rich set of existing libraries for many aspects facilitate the development of the tool as a whole. Thus Java is both: a technology that lends itself to a number of elegant reengineering techiques as well as a powerful means to implement a reengineering tool. The former provide access to the necessary tracing information while the latter processes this information.

The tool described in this paper is currently being integrated with the MORABIT component runtime infrastructure, a middleware for resource-aware runtime testing [43]. In the future we plan to enhance our own prototype implementation to include advanced features such as animation export, multithreading support and plug-in-based IDE and API-based custom integrations.

7. ACKNOWLEDGMENTS

This work has been funded by the Klaus Tschira Foundation (KTS) and the Landesstiftung Baden-Württemberg within the MORABIT research project. We thank our colleagues Elad Messing and Daniel Brenner for reviewing the manuscript and the anonymous reviewers for providing helpful suggestions for improvement.

8. REFERENCES

[1] Adobe Systems. *PDF Reference Fifth Edition, Adobe Portable Document Format Version 1.6*. Adobe System Inc., partners.adobe.com, 2004.

[2] Apache. *Velocity*. The Apache Jakarta Project, jakarta.apache.org/velocity/.

[3] Apache. *Batik SVG Toolkit*. The Apache XML Project, xml.apache.org/batik/.

[4] Binder, R. *Testing Object Oriented Systems. Models, Patterns and Tools*. Addison Wesley, 1999.

[5] Briand, L.C., Labiche, Y., and Leduc, J. *Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Real-Time Java Software*. Technical Report SCE-04-04, Carleton University, 2004.

[6] Briand, L. C., Labiche, Y., and Miao, Y. Towards the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering* (November 13 - 17, 2003). WCRE. IEEE Computer Society, Washington, DC, 2003, 57.

[7] Burd, E., Overy, D., and Wheetman, A. Evaluating Using Animation to Improve Understanding of Sequence Diagrams. In *Proceedings of the 10th international Workshop on Program Comprehension* (June 27 - 29, 2002). IWPC. IEEE Computer Society, Washington, DC, 2002, 107.

[8] ECMA International: *Standard ECMA-262 ECMAScript Language Specification*. ECMA International, www.ecma-international.org, 1999.

[9] Eclipse Project: *The EclipseUML2 project*. Eclipse Project Universal Tool Platform, www.eclipse.org/uml2/.

[10] Eclipse project. *Eclipse Modeling Framework (EMF)*. Eclipse Project Universal Tool Platform, www.eclipse.org/emf/.

[11] Eclipse Project: *Graphical Editing Framework (GEF)*. Eclipse Project Universal Tool Platform, www.eclipse.org/gef/.

[12] Eclipse Project: *AspectJ project*. eclipse.org, www.eclipse.org/aspectj/.

[13] Filman, R. E., Elrad, T., Clarke, S. and Aksit, M. *Aspect-Oriented Software Development*. Pearson Education, 2005.

[14] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Gradecki, J. and Lesiecki, N. *Mastering AspectJ - Aspect Oriented Programming in Java*. Wiley Publishing Inc, 2003.

[16] Gargiulo, J. and Mancoridis, S. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering SEKE* (June 2001). 2001.

[17] Gannod, G. and Murthy, S. Using Log Files to Reconstruct State-Based Software Architectures. In *Proceedings of the Working Conference on Software Architecture Reconstruction Workshop*. IEEE, 2002, 5-7.

[18] GNU: *The plotutils Package*. Free Software Foundation Inc., www.gnu.org/software/plotutils/.

[19] Greenfield, J., Short, K., Cook, S. and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing Inc, 2004.

[20] Hamou-Lhadj, A. and Lethbridge, T. C. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre For Advanced Studies on Collaborative Research* (Markham, Ontario, Canada, October 04 - 07, 2004). H. Lutfiyya, J. Singer, and D. A. Stewart, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 2004, 42-55.

[21] Jacobsen, I., Griss, M. and Jonnson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997

[22] JGraph ltd. *JGraph*. www.jgraph.com/.

[23] Kiselev, I. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, 2003.

[24] Fraikin, F. and Leonhardt, T. SeDiTeC " Testing Based on Sequence Diagrams. In *Proceedings of the 17th IEEE international Conference on Automated Software Engineering* (September 23 - 27, 2002). Automated Software Engineering. IEEE Computer Society, Washington, DC, 2002, 261.

[25] Kollman, R., Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (Wcre'02)* (October 29 - November 01, 2002). WCRE. IEEE Computer Society, Washington, DC, 2002, 22.

[26] Krasner, G. E. and Pope, S. T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. SIGS Publications, 1988, pages 26-49.

[27] Lowagie, B. *iText*. www.lowagie.com/iText/.

[28] Mukerji, J. and Miller, J. *MDA Guide Version 1.0.1*. Object Management Group, www.omg.org, 2003.

[29] Merdes, M., Häußler, J. and Zipf, A. GML2GML: Generic and Interoperable Round-Trip Geodata Editing - Concepts and Example. *8th AGILE Conference on GIScience*, 2005.

[30] OMG: *Meta Object Facility (MOF) Specification Version 1.4*. Object Management Group, www.omg.org, 2002.

[31] OMG: *XML Metadata Interchange (XMI) Specification version 2.0*. Object Management Group, www.omg.org, 2003.

[32] OMG: *UML 2.0 Superstructure Specification*. Object Management Group, www.omg.org, 2004.

[33] Oechsle, R. and Schmitt, T. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the

Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, international Seminar* (May 20 - 25, 2001). S. Diehl, Ed. Lecture Notes In Computer Science, vol. 2269. Springer-Verlag, London, 2002, 176-190.

[34] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, international Seminar* (May 20 - 25, 2001). S. Diehl, Ed. Lecture Notes In Computer Science, vol. 2269. Springer-Verlag, London, 2002, 151-162.

[35] Qoppa. *jPDFWriter*. Qoppa Software, www.qoppa.com/jpindex.html.

[36] PRESTO. *RED Project*. Presto Research Group Ohio State University, nomad.cse.ohio-state.edu/red/.

[37] Rountev, A., Volgin, O., and Reddoch, M. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal, September 05 - 06, 2005). M. Ernst and T. Jensen, Eds. PASTE '05. ACM Press, New York, NY, 2005, 96-102.

[38] Systä, T., Koskimies, K., and Müller, H. 2001. Shimba—an environment for reverse engineering Java software systems. *Softw. Pract. Exper.* 31, 4 (Apr. 2001), 371-394.

[39] Soley, R. and Group, O. S. *Model Driven Architecture*. Object Management Group, www.omg.org, 2000.

[40] Spinellis, D. *UMLGraph*. www.spinellis.gr/sw/umlgraph/.

[41] Spinnelis, D.: On the Declarative Specification of Models. *IEEE Software Volume 20 Issue 2*. 2003, pages 94-96.

[42] Sharp, R. and Rountev, A. Interactive Exploration of UML Sequence Diagrams. In *Proceedings of the IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*. 2005, 8-13.

[43] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R. The MORABIT Approach to Runtime Component Testing. In *Proceedings of the Second International Workshop on Testing and Quality Assurance for Component-Based Systems. (TQACBS06)*. 2006

[44] Sun Microsystems. *API Enhancements to the JavaBeans Component API in v1.4*. Sun Microsystems Inc, java.sun.com/j2se/1.4.2/docs/guide/beans/changes14.html, 2002.

[45] Sun Microsystems. *Java Platform Debugger Architecture (JPDA)*. Sun Microsystems Inc., http://java.sun.com/products/jpda/index.jsp.

[46] Stahl, T. and Völter, M. *Model-Driven Software Development*. Wiley, 2006.

[47] Sysoft. *Animation of UML Sequence Diagrams" - Amarcos*. Sysoft, http://www.sysoft-fr.com/en/Amarcos/ams-uml.asp.

[48] Thoms, C. and Holzer, B. Codegenerierung mit dem openArchitectureWare Generator 3.0 - The Next Generation. *javamagazin 07/2005*, 2005.

[49] W3C. *Extensible Stylesheet Language (XSL) Version 1.0*. W3C Recommendation, www.w3.org, 2001.

[50] W3C. *Scalable Vector Graphics (SVG) Version 1.1 Specification*, W3C Recommendation, www.w3.org, 2003

[51] W3C. *Java Language Binding for the SVG Document Object Model*. W3C Recommendation, www.w3.org. 2003.

[52] W3C. *ECMAScript Language Binding for SVG, W3C Recommendation*, www.w3.org, 2003.

[53] W3C. *XSL Transformations (XSLT) Version 1.0 Specification*. W3C Recommendation, www.w3.org, 1999

Propagation of JML non-null annotations in Java programs

Maciej Cielecki Jędrzej Fulara

Krzysztof Jakubczyk Łukasz Jancewicz

Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warszawa, Poland

{m.cielecki,j.fulara,k.jakubczyk,l.jancewicz}@students.mimuw.edu.pl

ABSTRACT

Development of high quality code is notably difficult. Tools that help maintaining the proper quality of code produced by programmers can be very useful: they may increase the quality of produced software and help managers to ensure that the product is ready for the market. One of such tools is ESC/Java2, a static checker of Java Modeling Language annotations. These annotations can be used to ensure that a certain assertion is satisfied during the execution of the program, among the others - to ensure that a certain variable never has a null value. Unfortunately, using ESC/Java2 can be very troublesome and time-consuming for programmers, as it lacks a friendly user interface and a function that propagates annotations.

We present *CANAPA*, a tool that can highly reduce time and effort of eliminating null pointer exceptions in Java code. This tool can automatically propagate JML non-null annotations and comes with a handy Eclipse plug-in. We believe that functionality of *CANAPA* will minimize the effort required to benefit from using the JML non-null checking.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*Software/Program Verification*

General Terms

RELIABILITY, VERIFICATION

1. INTRODUCTION

1.1 Coding errors

Ensuring that a piece of software created within a company is free of coding bugs has always been a huge problem. Company managers try to succeed in this area by applying various coding policies and code-checking strategies. These policies can include overnight bug checking (people are hired to work at night and find errors in code that was created during the day), enforcing coding standards, like consisting naming convention or limiting length of methods, using automated tools to verify certain properties of the code either by static code analysis or dynamic assertion checking. The efficiency of the tools of the latter category can usually be improved by annotating the code with some meta information, like pre- and post- condition of methods or invariants of data structure implementations.

There are many languages created to annotate programs, but none of them is actually very popular. For the Java programming language the de facto standard is JML, the Java Modeling Language [11, 3]. Recently Microsoft introduced Spec# [2], the extension of C# targeted at specifying correctness properties of code. Some work has also been done for low-level languages. In particular, the Java Bytecode has its own specification language, called BCSL or BML [4]. The motivation for the latter was ensuring the security of Java applets on SmartCards.

In general, coding errors can be divided into two categories: those that result from a programmer's misunderstanding of the algorithm (for example, adding two variables instead of multiplying) and those that result from a programmer's carelessness (like leaving a variable uninitialized).

Although it is very hard to detect the first kind of bugs, there is a way to avoid a large majority of the second ones, mainly by creating and using automated verification software. It is of course impossible to write a tool that automatically checks the correctness of all programs, but there is a way to check some of its features.

1.2 Avoiding null pointer exceptions in Java

The most common error found in pointer-based object-oriented programming languages occurs when one tries to access an object variable that has not been initialized. In Java, it is the well-known null pointer exception. Null pointer exceptions are a very serious threat to the safety of programs, because when they occur at run-time, they cause a total software failure. That is why it is important to support the programmer in detecting and eliminating these kinds of problems. But first, the programmer has to express what features he would expect from his or her software. For this task we use JML, a behavioral interface specification language. One of the key features of JML is the possibility to annotate certain variables as non-null, which means that it was the programmer's intention not to allow to assign a null value to that variable.

1.3 JML

1.3.1 Overview

JML, the Java Modeling Language, is useful for specifying detailed design properties of Java classes and interfaces. JML is a behavioral interface specification language for Java. The behavior of classes and method created by a programmer can be precisely documented in JML annotations. They describe the intended way that programmers should use the code. JML can, for example, list the preconditions and postconditions for methods as well as class invariants.

An important goal for the design of the JML specification language is that it should be easily understandable by Java programmers. It is achieved by staying as close as possible to Java syntax and semantics. Several groups worldwide are now building tools that support the JML notation and are involved with the ongoing design of JML. The open, cooperative nature of the JML effort is important both for tool developers and users. For potential users, the fact that there are several tools supporting the same notation is clearly an advantage. For tool developers, using a common syntax and semantics can make it much easier to get users interested, because one of the biggest problem with starting to use a new specification tool is often the lack of familiarity with the specification language.

1.3.2 Non_null annotations

In this paper, we focus on annotating properties of methods, variables etc. which assure that the object under question never has a null value.

In JML, there are two ways to make such an assertion. If we want to make sure that a variable is never null (for example, we would call its method in a moment and it could produce a null pointer exception), we add the `/*@ non_null @*/` annotation (note the `@` sign after the beginning and before the end of the comment):

```
/*@ non_null @*/ String s = "Hi there!";
```

A more interesting example is the method definition. If we want a method argument to be non-null, we could write something like this:

```
public void checkLength(/*@ non_null @*/String s);
```

or, we could add something like that:

```
//requires s != null
public void checkLength(String s);
```

Notice the subtle difference between those examples. In the first one, if the method body would contain the line:

```
s = null;
```

we would get an error. In the second example, as long as at entry point the non-null assertion is fulfilled, the statement won't generate an error. By the way, we find it a bad programming practice to change parameters that way, they should be copied to another variable instead.

1.3.3 JML checking

An annotation language like JML would be quite useless without a tool that can extract information from the annotations and use it to verify some, if not all, of its required features. In general, we divide the checkers into two categories:

- run-time checking tools, like JMLrac [5] — annotations are converted into assertions that are verified when the code they describe is executed
- static checking tools, like ESC/Java and ESC/Java2 [9] — do not require running the program; instead they try to prove that annotations are fulfilled by statically analysing possible execution paths.

Advantages and disadvantages of each method can be clearly seen. Run-time checkers can check any assertion, no matter how complicated, but if a method is never run, its assertions will not be executed and verified. Besides, the execution time is longer due to additional instructions in the code. Static checkers, on the other hand, are limited by their reasoning capabilities. Hence they can sometimes show nonexistent errors (false positives) or fail to find some existing ones (false negatives). The most popular static checker for Java is ESC/Java2 [9].

1.4 ESC/Java2

ESC/Java tool, developed at Compaq Research, performs what is called extended static checking, a compile-time check that goes well beyond type checking. It can check relatively simple assertions and can check for certain kinds of common errors in Java code, such as dereferencing null, indexing an array outside its bounds, or casting a reference to an impermissible type. ESC/Java supports a subset of JML. ESC/Java2 [9] is an extension to ESC/Java, whose development has ended.

The user's interaction with ESC/Java2 is quite similar to the interaction with the compiler's type checker: the user includes JML annotations in the code and runs the tool, and the tool responds with a list of possible errors in the program. The use of JML annotations enables ESC/Java2 to produce warnings not at the source locations where errors manifest themselves at run-time, but at the source locations where the errors are committed.

The creators of ESC/Java2 wanted it to be as fast as possible, even at the cost of soundness and completeness. ESC/Java2 translates a given JML-annotated Java program into verification conditions, logical formulas that are valid if and only if the program is free of the kinds of errors being analyzed. The verification conditions are fed to an automatic first-order theorem prover Simplify [7], which tries to prove them. Any verification-condition counterexamples found by Simplify are turned into programmer-sensible warning messages, including the kind and source location of each potential error.

2. ANNOTATING THE PROGRAM

The combined usage of JML non-null annotations and ESC/Java2 allows software developers to eliminate all null pointer exceptions from their programs. However, benefits of doing so do not always compensate additional time spent on manually adding the necessary assertions. Several add-on tools were developed to make the process faster, such as the ESCJava2 Eclipse plug-in that highlights places of possible errors. Unfortunately, that is still not enough to convince programmers to use JML.

Using ESC/Java2 to check null pointer exceptions is somewhat cumbersome, because the checker shows us the place in the code where the error might occur, but it does not tell us where to put the `/*@ non_null @*/` annotation. Very often the need of inserting another annotation is so obvious, that we would expect it to be done for us.

Let's consider the following example:

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "eLLo";
 }
 void set(String param) {
  attribute = param;
     //ESCJava2 will point to this line
 }
}
```

ESC/Java2 will signal an error in the assignment inside the method `set()`, so the programmer should add a non-null annotation to the parameter `param`. Then one can discover that, for example, one of `set()` method calls take a parameter which is not annotated non-null. The programmer is forced to correct his or her code and run ESC/Java2 each time he or she does it until all the errors are eliminated.

Other examples of cases when the annotation should and should not be propagated can be found in Section5.

An obvious solution is to create a tool that supports the programmer in annotating his or her code. We would expect the following features from such a tool:

- it should propagate annotations inserted by the user to avoid pointless ESC/Java2 warnings
- it should be fairly easy to use
- it should not require additional annotations in the code to make it work
- it should propagate only those annotations for which it is obvious that they should be propagated
- it should integrate into a popular Java development platform
- its effects should be easily reversible

3. RELATED TOOLS

Our solution, *CANAPA*, is based on the Java Modeling Language and ESC/Java2. There exist several other languages and systems that aim at statical enforcement of program correctness.

In Visual Studio 2005, Microsoft introduced Code Analysis tools [6]. Among other features, these tools can check the program for potential null-pointer dereference errors. There is no support for Java, but one can annotate C++ code or write full specifications of C# programs in Spec#.

There are many static checkers for C language that can check the NULL values to some extent. Many of those are commercial, closed source software, therefore are not broadly available. Nevertheless some of those checkers are very powerful, designed for large codebases, support user defined properties, with very small number of false positives. They are also usually bundled with an entire package of tools that enforce code quality, see eg. [14].

There were many research about the subject of annotations [4], [13]. There is considerable interest in automated annotation propagation, but the approaches considered were different from ours.

There are various tools that were build around ESC/Java. Two most interesting from our point of view are: The Daikon Invariant Detector and The Houdini Tool.

3.1 Daikon Invariant Detector

Daikon [8] is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports properties that hold at a certain point or points in a program. Daikon runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. It can be used to automatically generate JML annotations in Java Code.

3.2 Houdini

This tool was under development as a research project at Compaq SRC. Houdini infers ESC/Java annotations for a given program and then runs ESC/Java to obtain a set of warnings. This set is considerably smaller, and more interesting, than the set of mostly spurious warnings that ESC/Java would have produced on the unannotated program. Although this process does not provide the same benefit of documented programmer design decisions (it detects the de facto design rather than enforcing de juro design decisions), Houdini greatly reduces the cost of finding common errors in a given program. Non-null annotations are among the annotations generated by Houdini, but the approach taken by the creators of this software is different from ours and does not guarantee that full set of annotations will be generated.

4. OUR SOLUTION

We present *CANAPA*, ”Completely Automated Non-null Annotation Propagating Application”, a tool to automatically propagate JML annotations that concern being or not being null by variables, method result etc. *CANAPA* is a program that propagates the `/*@ non_null @*/` annotations inside the source code ”bottom-up”. This greatly reduces time and effort to correctly insert non-null annotations into the code.

4.1 Overview

The main idea is that the programmer inserts a JML non-null annotation inside the code, and CANAPA checks what are the obvious implications of such an insertion and inserts additional non-null annotations where ESC/Java2 would expect them. This way, the programmer sees the error in his logic (if any) at its source, and does not have to manually add each assertion to get to the mistake.

CANAPA has the following features:

- the program (*CANAPA*) is idempotent - the result of running it once on a Java code should be the same as running it twice
- from the preceding, it cannot add non-null annotations to class attributes - this would lead to undesirable results - see Use cases
- it changes the code only when it's sure it was the programmer's intention
- it adds its own comment tag to the JML tag in case the programmer wanted to remove the effects of its work

The Usage of CANAPA is fairly straightforward. You simply put `/*@ non_null @*/` annotations in the code where you want them and then run our tool, which propagates those assertions anywhere it is necessary.

CANAPA can be invoked from the command line with a directory parameter, or executed from Eclipse development platform via a plugin. During its work, *CANAPA* adds annotations to selected files and creates their backups, notifying the user which files were modified in the process.

Each annotation added by *CANAPA* is marked with a `/*CANAPA*/` prefix. Annotations existing before running the tool will not have this marker. This way, you can easily find and, if necessary, remove the automatically added annotations.

CANAPA comes with a handy Eclipse plug-in that allows to run it within the Eclipse programming environment. The tool simply adds annotations to the file looked at by the programmer and the text output can be seen in the Console window.

4.2 Implementation Details

The tool consists of a number of elements: the interface to ESC/Java2 (to find errors), a Java code parser to insert needed annotations, a simple text user interface and the Eclipse plug-in.

ESC/Java2 is invoked directly via its main method from the JAR, that's why our software requires Java 1.4 to work.

The parser used in the tool is a slightly modified free JParse [12] tool, which itself is based on ANTLR [1], a free parser generator.

The algorithm used in the tool is as follows: ESC/Java2 is run on the code provided by the user. The errors returned by ESC/Java2 are parsed, their solutions (if any) found and appropriate annotations placed to remove ESC/Java2 errors. Then ESC/Java2 is run again (this time it won't detect errors where they were before). If any new errors are detected, the procedure is repeated.

The number of iterations of the algorithm is limited by the depth of the deepest variable and method call dependency in the user's code. It must be noted that, with a "clean" (unannotated) large piece of code running *CANAPA* for the first time may take some time. However, the more annotations are in the code, the faster our tool is. In the ideal working example, when the programmer starts annotating his or her code from the very beginning and runs the tool each time he or she makes a significant addition, *CANAPA* will work very fast, with few iterations.

CANAPA tries to correct the following ESC/Java2 errors:

- assignment of a possibly null item to a non-null annotated variable:
  - assignment of a method parameter
  - assignment of a local variable
  - assignment of a function result
- dereferencing a possibly null item:
  - invoking `variable.someMethod()`
  - invoking `oneMethod().anotherMethod()`

The action taken by *CANAPA* differs depending on the type of item in question:

- if the item is a local variable, annotate its declaration with `/*@ non_null @*/`
- if the item is a formal method parameter, annotate it in the method header with `/*@ non_null @*/`
- if the item is a result of a method, annotate the return type of the method in the method header with `/*@ non_null @*/`
- if the item is a class attribute, do not annotate it - this probably would not be what the programmer wants, as it could cause "top-down" propagation into other methods

It is significant that the tool does not modify the code itself, but only the comments. So the compiler would still work if something went wrong.

The tool does its best to propagate the annotations just as the programmer would. There is, however, one situation in which *CANAPA* fails to predict the right annotation. Let us imagine that the programmer writes a method without annotating its parameter and dereferences it in the method body. It is impossible to know whether the intention of the programmer was to never call this method with the null argument or he simply forgot to put the if clause. Since *CANAPA* cannot guess what to write in the else branch, anyway, it annotates the parameter with non-null.

Fortunately, there is a way to deal with the situation. The Eclipse plug-in provides an option to revert the effects of the last *CANAPA* corrections within a few keystrokes. To avoid programmer confusion about which changes were added in the last *CANAPA* execution, a "commit" option is added that eliminates the `/*CANAPA*/` comments before `/*@ non_null @*/` annotations.

5. USE CASES

In this chapter we will show several basic examples of using *CANAPA*. Each example contains of a short piece of incorrect code and the description how *CANAPA* deals with it.

5.1 Example 1

This example shows how a `/*@ non_null @*/` annotation can be propagated to a method parameter.

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "Attribute";
 }
 void set(String param) {
  attribute = param;
 }
}
```

The code presented above is not correct: ESC/Java2 will point to the line `attribute = param`. *Attribute* is declared as non-null, and we try to assign *param* to it, so *param* must be declared as `/*@ non_null @*/` too. The easiest way to correct it is to add a `/*@ non_null @*/` annotation to *param* in the method header. Launching *CANAPA* will modify the code as follows:

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "Attribute";
 }
 void set(/*CANAPA*//*@non_null@*/ String param) {
  attribute = param;
 }
}
```

And that is exactly what our application does. There is another possible way to correct this error - it involves enclosing the assignment in if-else statement. However, it is impossible for the tool to guess what to do if param is null.

5.2 Example 2

This example shows the inference of a `/*@ non_null @*/` annotation to a variable or parameter, of which a programmer invokes a method. Let's consider the following piece of code:

```
class ClassA {
 public A(){}
 public void methodA(){}
}
class ClassB {
 public B(){}
 public void methodB(ClassA a){
  a.methodA();
 }
}
```

This code is invalid, as the parameter `a` of `methodB` could be null. So the method call `a.methodA()` may cause a null pointer exception. To correct the error, one should add a `/*@ non_null @*/` annotation to the parameter in the methodB header. After launching our application, the code will be modified as follows:

```
class ClassA {
 public A(){}
 public void methodA(){}
}
class ClassB {
 public B(){}
 public void
        methodB(/*CANAPA*//*@non_null@*/ClassA a){
  a.methodA();
 }
}
```

Of course the problem concerns not only parameters, but also variables:

```
class Class{
 public Class(){}
 public void method(){
  String str;
  str.substring(1);
 }
}
```

The local variable `str` cannot be null, otherwise the method call `str.substring(1)` would cause a null pointer exception. The solution is to declare `str` as `/*@ non_null @*/`. *CANAPA* will add the appropriate annotation. Of course, the problem persists (`str` is uninitialized), but this time, ESC/Java2 error points the user exactly to the source of the problem.

5.3 Example 3

This example shows how a `/*@ non_null @*/` annotation can be propagated to the method's result.

```
class Class{
 /*@ non_null @*/ String attribute;
 public Class(){
  attribute = "Attribute";
 }
 private String getString(){
  return "This is a string";
 }
 public void set(){
  attribute = getString();
 }
}
```

$^{a}_{b}$ We assign the result of `getString()` to `attribute`, which is declared as `/*@ non_null @*/`. Until we are not sure that the method `getString()` cannot return a null, this code will be incorrect. The easiest way to solve this problem is to add `/*@ non_null @*/` annotation to the result of `getString()`. The code modified by *CANAPA* will be as follows:

```
class Class{
 /*@ non_null @*/ String attribute;
 public Class(){
  attribute = "Attribute";
 }
 private /*@ non_null @*/ String getString(){
  return "This is a string";
 }
 public void set(){
  attribute = getString();
 }
}
```

5.4 Example 4

This example shows that there are situations, when a propagation should not be done, although one could think that an annotation should be added. Consider following piece of code:

```
class Class{
 String attribute;
 void doSomething(){
  ...
  /*@non_null@*/String str = attribute;
 }
 void setNull(){
  attribute = null;
 }
}
```

One might expect that *CANAPA* will add an annotation to the attribute, modifying the code as follows:

```
class Class{
 /*@non_null@*/ String attribute;
 void doSomething(){
  ...
  /*@non_null@*/ String str = attribute;
 }
 void setNull(){
  attribute = null;
 }
}
```

After careful consideration of this code, we can see that the added annotation causes an error in an other method. Namely in the `setNull()` method (we will try to assign null to an attribute that was declared as non-null). This definitely would not be acceptable for most of programmers. Moreover, we claim that in such a situation it is impossible to modify the code automatically in a reasonable way. So we have decided not to add anything to class attributes.

6. SUMMARY

We created *CANAPA*, the tool that highly reduces time and effort of eliminating null pointer exceptions in Java code. This tool automatically propagates JML non-null annotations, whenever this results from the programmer's intension. It also comes with a handy Eclipse plug-in to increase productivity. *CANAPA* is distributed under the GNU LESSER GENERAL PUBLIC LICENSE [10]. It is available from `http://www.mimuw.edu.pl/~chrzaszcz/Canapa/`. It requires a Java Runtime Environment (version 1.4) and the ESC/Java2 checker. To run the CANAPA Eclipse plug-in, version 3.1 or higher of the Eclipse environment is needed.

7. ACKNOWLEDGEMENT

This work was partly supported by the Information Society Technologies programme of the European Commission, under the IST-2005-015905 MOBIUS project and Sixth Framework Marie-Curie Programme MEIF-CT-2005-024306 SOJOURN. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

8. ADDITIONAL AUTHORS

Jacek Chrząszcz, Institute of Informatics, Warsaw University, email: `chrzaszcz@mimuw.edu.pl`.

Aleksy Schubert, Institute of Informatics, Warsaw University, Poland and SoS Group, Faculty of Science, University of Nijmegen, email: `alx@mimuw.edu.pl`.

Łukasz Kamiński, Comarch Research and Development Center, email: `Lukasz.Kaminski@comarch.pl`.

9. REFERENCES

[1] Antlr parser generator. http://www.antlr.org/.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. In T. Arts and W. Fokkink, editors, *FMICS: Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, June 5-7 2003.

[4] Lilian Burdy and Mariela Pavlova. Java bytecode specification and verification. In *21st Annual ACM Symposium on Applied Computing (SAC'06)*, Dijon, Apr 2006. ACM Press.

[5] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[6] Code analysis for C/C++ — overview. http://msdn2.microsoft.com/en-us/library/d3bbz7tz.aspx.

[7] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[9] Extended Static Checker for Java version 2. http://secure.ucd.ie/products/opensource/ESCJava2/.

[10] GNU LESSER GENERAL PUBLIC LICENSE. http://www.gnu.org/copyleft/lesser.html.

[11] The Java Modeling Language (JML). http://www.cs.iastate.edu/ leavens/JML//index.shtml.

[12] JParse: a Java parser. http://www.ittc.ku.edu/JParse/.

[13] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*, Toulouse, France, August 2004. Kluwer Academic Publishers.

[14] Static source code analysis tools for C. http://www.spinroot.com/static/.

Session F
Novel Uses of Java

On the Design of a Java Computer Algebra System

Heinz Kredel
IT-Center
University of Mannheim
Mannheim, Germany
kredel@rz.uni-mannheim.de

ABSTRACT

This paper considers Java as an implementation language for a starting part of a computer algebra library. It describes a design of basic arithmetic and multivariate polynomial interfaces and classes which are then employed in advanced parallel and distributed Groebner base algorithms and applications. The library is type-safe due to its design with Java's generic type parameters and thread-safe using Java's concurrent programming facilities.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures; G.4 [**Mathematical Software**]: Computer Algebra; I.1 [**Symbolic and Algebraic Manipulation**]: Specialpurpose algebraic systems

General Terms

Design, Algorithms, Type-safe, Thread-safe

Keywords

computer algebra library, multivariate polynomials

1. INTRODUCTION

We describe an object oriented design of a Java Computer Algebra System (called JAS in the following) as type safe and thread safe approach to computer algebra. JAS provides a well designed software library using generic types for algebraic computations implemented in the Java programming language. The library can be used as any other Java software package or it can be used interactively or interpreted through an jython (Java Python) front end. The focus of JAS is at the moment on commutative and solvable polynomials, Groebner bases and applications. By the use of Java as implementation language JAS is 64-bit and multi-core cpu ready. JAS is developed since 2000 (see the weblog in [12]) and was partly described in [11].

Recall form mathematics that a multivariate polynomial p is an element of a polynomial ring R in n variables over some coefficient ring C, i.e. in formal notation $p \in R = C[x_1, \ldots, x_n]$, e.g.

$$p = 3x_1^2 x_3^4 + 7x_2^5 - 61 \ \in \ \mathbb{Z}[x_1, x_2, x_3]$$

is a polynomial in 3 variables over the integers. Note, that the definition is recursive in the sense, that C can be another polynomial ring. More formally a polynomial is a mapping from a monoid T to a ring C, $p = T \longrightarrow C$ where only finitely many elements of T are mapped to non-zero elements of C. In case $R = C[x_1, \ldots, x_n]$ the monoid T, is generated by terms, i.e. (commutative) products of the variables $x_i, i = 1, \ldots, n$. In our example the map is $p =$

$$x_1^2 x_3^4 \mapsto 3, x_2^5 \mapsto 7, x_1^0 x_2^0 x_3^0 \mapsto -61, \text{ else } x_1^{e_1} x_2^{e_2} x_3^{e_3} \mapsto 0.$$

This view is used to implement a polynomial using a `Map` from the Java collection framework. A computer representation of an element of T is used as key and the value is a representation of a (non-zero) element of C, i.e. keys being mapped to zero are not stored in the `Map`. Since some properties of multivariate polynomial rings depend on a certain ordering $<_T$ on the monoid T we actually use a `SortedMap` (with the `TreeMap` implementation). The ordering $<_T$ determines e.g. which monoid in the polynomial is the highest, just as the usual degree does for univariate polynomials. Addition and multiplication of polynomials is defined as usual, the zero polynomial is the empty map and the one polynomial is the map $x_1^0 x_2^0 \ldots x_n^0 \mapsto 1$, where 1 denotes the representation of the one of C. We also consider solvable polynomials which are multivariate polynomials with commutative addition and a non-commutative multiplication $*$ with respect to relations

$$x_j * x_i = c_{ij} x_i x_j + p_{ij},$$

for $1 \leq i < j \leq n, 0 \neq c_{ij} \in C, x_i x_j >_T p_{ij} \in R$. The (mathematical) class of solvable polynomial rings naturally contains the class of polynomial rings. So a polynomial is always a solvable polynomial with respect to commuting relations. One may then think that a polynomial class (in computer science sense) could extend a solvable polynomial class but it is the other way round.

Based on this sketch of polynomial mathematics the paper describes the basic arithmetic and multivariate polynomial part of a bigger library, which consists of the following additional packages. The package `edu.jas.ring` contains classes for polynomial and solvable polynomial reduction, Groebner bases and ideal arithmetic as well as thread parallel and distributed versions of Buchbergers algorithm like ReductionSeq, GroebnerBaseAbstract, GroebnerBaseSeq, GroebnerBaseParallel and GroebnerBaseDistributed. The package `edu.jas.module` contains classes for module Groebner bases and syzygies over polynomials and solvable polynomials like ModGroebnerBase or SolvableSyzygy. Finally `edu.jas.application` contains classes with applications of Groebner bases such as ideal intersections and ideal quotients implemented in Ideal or SolvableIdeal.

1.1 Related Work

An overview of computer algebra systems and also on design issues can by found in the "Computer Algebra Handbook" [8]. For the scope of this paper the following work was most influential: Axiom [10] and Aldor [20] with their comprehensive type library and category and domain concepts. Sum-It [4] is a type safe library based on Axiom and Weyl [22] presents a concept of an object oriented computer algebra library in Common Lisp. Type systems for computer algebra are proposed by Santas [19] and existing type systems are analyzed by Poll and Thomson [18]. Other library implementations of computer algebra are e.g. LiDIA [5] and Singular [9] in C++ and MAS [13] in Modula-2.

Java for symbolic computation is discussed in [3] with the conclusion, that it fulfills most of the conceptional requirements defined by the authors, but is not suitable because of performance issues (Java up to JDK 1.2 studied). In [1] a package for symbolic integration in Java is presented. A type unsafe algebraic system with Axiom like coercion facilities is presented in [6]. A computer algebra library with maximal use of patterns (object creational patterns, storage abstraction patterns and coercion patterns) are presented by Niculescu [15, 16]. A Java API for univariate polynomials employing a facade pattern to encapsulate different implementations is discussed in [21]. An interesting project is the Orbital library [17], which provides algorithms from (mathematical) logic, polynomial arithmetic with Groebner bases and genetic optimization.

Due to limited space we have not discussed the related mathematical work on solvable polynomials and Groebner base algorithms, see e.g. [2, 7] for some introduction.

1.2 Outline

In section 2 we present an overall view of the design of the central interfaces and classes. We show how part of the Axiom / Aldor basic type hierarchie can be realized. We discuss the usage of creational patterns, such as factory, abstract factory, or prototype in the construction of the library. We do currently not have an explicit storage abstraction and a conversion abstraction to coerce elements from one type to another. However the generic polynomial class `GenPolynomial` applies the facade pattern to hide the user form its complex internal workings. In section 3 we take a closer look at the functionality, i.e. the methods and attributes of the presented main classes. Section 4 treats some aspects of the implementation and discusses the usage of standard Java patterns to encapsulate different implementations of parallel Groebner base algorithms. Finally section 5 draws some conclusions and shows missing parts of the library.

2. DESIGN

One of the first things we have to decide is how we want

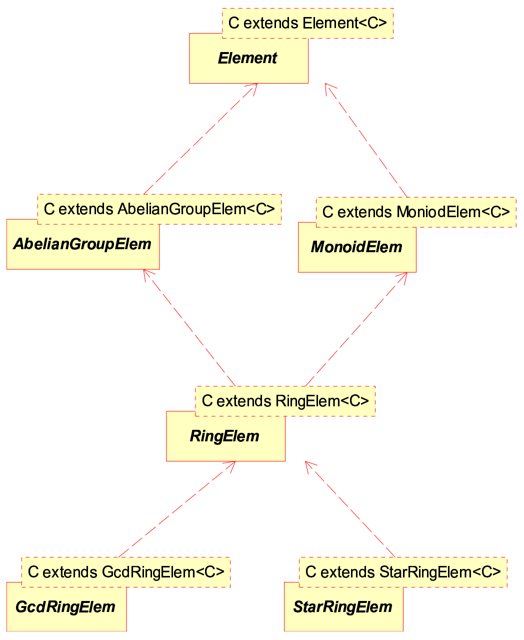


Figure 1: Overview of some algebraic types

to implement algebraic structures and elements of these algebraic structures. Alternatives are i) elements are implemented as (Java or C++) objects with data structure and methods or ii) elements are simple C++ like `structs` or records and algebraic structure functionality is implemented as (static) methods of module like classes. The *second* alternative is more natural to mathematicians, as they perceive algebraic structures as sets (of elements) and maps between such sets. In this view an algebraic structure is a collection of maps (or functions) and a natural implementation is as in FORTRAN as bunches of functions with elements (integers and floats) directly implemented by hardware types. However scientific function libraries implemented in this style are horrible because of the endless parameter lists and the endless repetitions of functions doing the same for other parameter types. The *first* alternative is the approach of computer scientists and it leads to better data encapsulation, context encapsulation and more modular and maintainable code. Since the algebraic elements we are interested in have sufficient internal structure (arbitrary precision integers and multivariate polynomials) we opt for encapsulation with its various software engineering advantages and so choose the first alternative. This reasoning also implies using Java as implementation language, since otherwise we could have used FORTRAN.

2.1 Type structure

Using Java generic types (types as parameters) it is not difficult to specify the interfaces for the most used algebraic types. The interfaces define a type parameter `C` which is required to extend the respective interface. The central interface is `RingElem` (see figures 1 and 3) which extends `AbelianGroupElem` with the additive methods and `MonoidElem` with the multiplicative methods. Both extend `Element` with methods needed by all types. `RingElem` is itself extended by `GcdRingElem` with greatest common divisor methods and `StarRingElem` with methods related to (complex) conjugation. This exemplifies the suitability of Java to implement Axiom / Aldor like type systems, although we do not present such a comprehensive type hierarchy as they do.

2.2 Ring element creation

Figure 2 (see also figure 3) gives an overview of the central classes. The interface `RingElem` defines a recursive type which defines the functionality (see next section) of the polynomial coefficients and is also implemented by the polynomials itself. So polynomials can be taken as coefficients for other polynomials, thus defining a recursive polynomial ring structure.

Since the construction of constant ring elements (e.g. zero and one) has been difficult in our previous designs, we separated the creational aspects of ring elements into ring factories with sufficient context information. The minimal factory functionality is defined by the interface `RingFactory`. Constructors for polynomial rings will then require factories for the coefficients so that the construction of polynomials over these coefficient rings poses no problem. The ring factories are additionally required because of the Java generic type design. I.e. if `C` is a generic type name it is not possible to construct a new object with `new C()`. Even if this would be possible, one can not specify constructor signatures in Java interfaces, e.g. to construct a one or zero constant ring element. Recursion is again achieved by using polynomial factories as coefficient factories in recursive polynomial rings. Constructors for polynomials will always require a polynomial factory parameter which knows all details about the polynomial ring under consideration.

2.3 Polynomials and coefficients

Basic coefficient classes, such as `BigRational` or `BigInteger`, implement both the `RingElem` and `RingFactory` interfaces. This is convenient, since these classes do not need further context information in the factory. In the implementation of the interfaces the type parameter `C extends RingElem<C>` is simultaneously bound to the respective class, e.g. `BigRational`. Coefficient objects can in most cases be created directly via the respective class constructors, but also via the factory methods. E.g. the object representing the number 2 can be created by `new BigRational(2)` or by `fac = new BigRational()`, `fac.fromInteger(2)` and the object representing the rational number 1/2 can be created by `new BigRational(1,2)` or by `fac.parse("1/2")`.

Generic polynomials are implemented in class `GenPolynomial`, which has a type parameter `C` which extends `RingElem<C>` for the coefficient type (see figures 2 and 3). So all operations on coefficients required in polynomial arithmetic and manipulation are guaranteed to exist by the `RingElem` interface. The constructors of the polynomials always require a matching polynomial factory. The generic polynomial factory is implemented in the class `GenPolynomialRing`, again with type parameter `C extends RingElem<C>` (not `RingFactory`). The polynomial factory however implements the interface `RingFactory<C extends RingElem<C>>` so that it can also be used as coefficient factory. The constructors for `GenPolynomialRing` require at least parameters for a coefficient factory and the number of variables of the polynomial ring.

Having generic polynomial and elementary coefficient implementations one can attempt to construct polynomial objects. The type is first created by binding the type parameter `C` to the desired coefficient type, e.g. `BigRational`. So we arrive at the type `GenPolynomial<BigRational>`. Polynomial objects are then created via the respective polynomial factory of type `GenPolynomialRing<BigRational>`, which is created by binding the generic coefficient type of the generic polynomial factory to the desired coefficient type, e.g. `BigRational`. A polynomial factory object is created from a coefficient factory object and the number of variables in the polynomial ring as usual with the new operator via one of its constructors. Given an object `coFac` of type `BigRational`, e.g. created with `new BigRational()`, a polynomial factory object `pf` of the above described type could be created by

```
new GenPolynomialRing<BigRational>(coFac,5).
```

I.e. we specify a polynomial ring with 5 variables over the rational numbers. A polynomial object `p` of the above described type can then be created by any method defined in `RingFactory`, e.g. by `pf.fromInteger(1)`, `pf.getONE()`, `pf.random(3)` or `pf.parse("1")`. See also the example in figure 4.

Since `GenPolynomial` itself implements the `RingElem` interface, they can also be used recursively as coefficients. We continue the polynomial example and are going to use polynomials over the rational numbers as coefficients of a new polynomial. The type is then

```
GenPolynomial<GenPolynomial<BigRational>>
```

and the polynomial factory has type

```
GenPolynomialRing<GenPolynomial<BigRational>>.
```

Using the polynomial coefficient factory `pf` from above a recursive polynomial factory `rfac` could be created by `new`

```
GenPolynomialRing<GenPolynomial<BigRational>>(pf,3)
```

The creation of a recursive polynomial object `r` of the above described type is then as a easy as before e.g. by `rfac.getONE()`, `rfac.fromInteger(1)` or `rfac.random(3)`.

2.4 Solvable polynomials

The generic polynomials are intended as super class for further types of polynomial rings. As one example we take solvable polynomials, which are like normal polynomials but are equipped with a new non-commutative multiplication. From mathematics one would expect that a polynomial class would extend a solvable polynomial class, but it it is the other way, since the multiplication method gets overwritten for non-commutative multiplication. The implementing class `GenSolvablePolynomial` extends `GenPolynomial` (see figures 2 and 3) and inherits all methods except `clone()` and `multiply()`. The class also has a type parameter `C` which extends `RingElem<C>` for the coefficient type. Note, that the inherited methods are in fact creating solvable polynomials since they employ the solvable polynomial factory for the creation of any new polynomials internally. Only the formal method return type is that of `GenPolynomial`, the run-time type is `GenSolvablePolynomial` to which they can be casted as required. The factory for solvable polynomials is implemented by the class `GenSolvablePolynomialRing` which also extends the generic polynomial factory. So this factory can also be used in the constructors of `GenPolynomial` to produce in fact solvable polynomials internally. The data

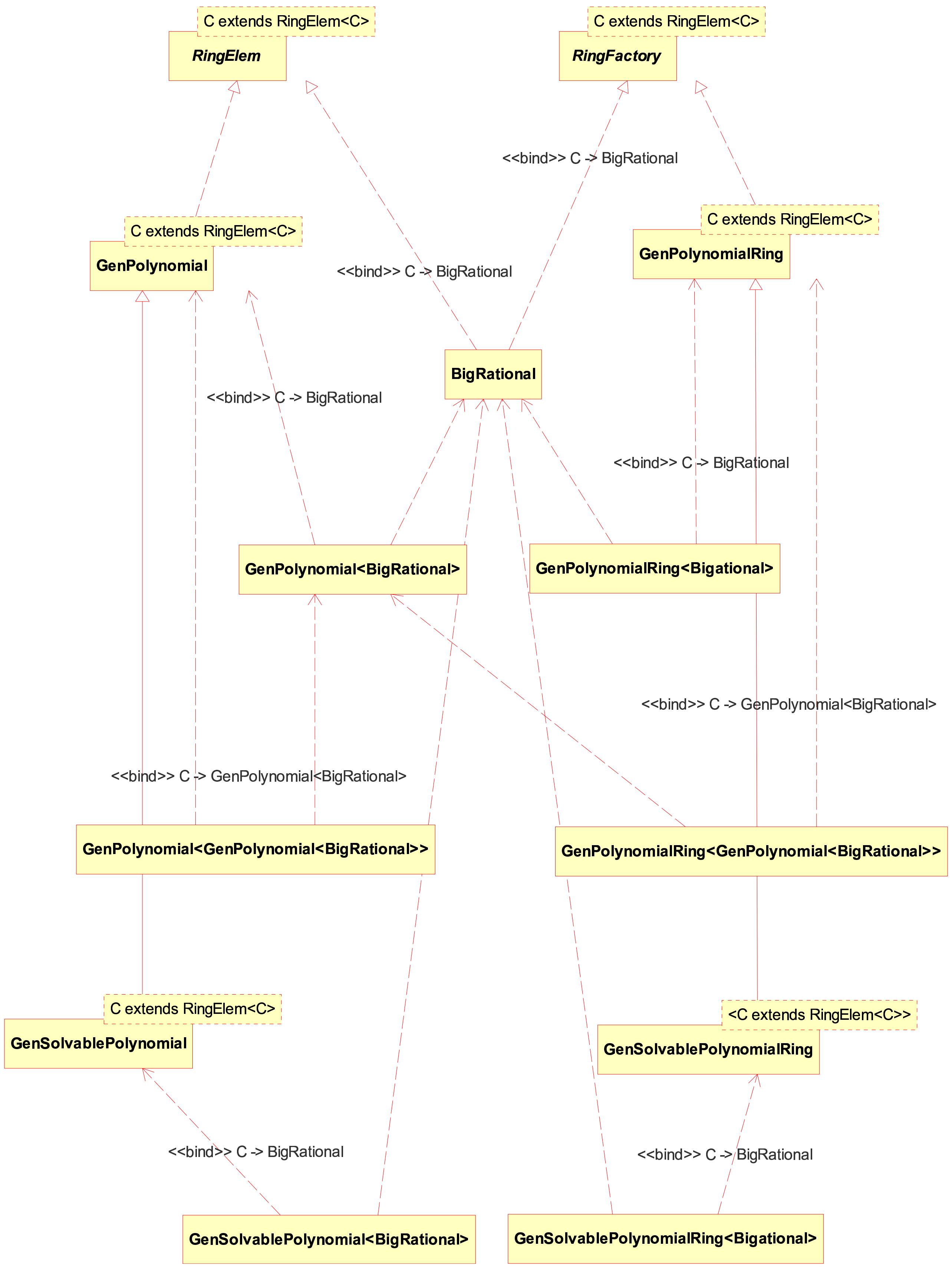

C extends RingElem<C>
RingElem
C extends RingElem<C>
RingFactory
<<bind>> C -> BigRational
C extends RingElem<C>
GenPolynomial
<<bind>> C -> BigRational
C extends RingElem<C>
GenPolynomialRing
BigRational
<<bind>> C -> BigRational
<<bind>> C -> BigRational
GenPolynomial<BigRational>
GenPolynomialRing<Bigational>
<<bind>> C -> GenPolynomial<BigRational>
<<bind>> C -> GenPolynomial<BigRational>
GenPolynomial<GenPolynomial<BigRational>>
GenPolynomialRing<GenPolynomial<BigRational>>
C extends RingElem<C>
GenSolvablePolynomial
<C extends RingElem<C>>
GenSolvablePolynomialRing
<<bind>> C -> BigRational
<<bind>> C -> BigRational
GenSolvablePolynomial<BigRational>
GenSolvablePolynomialRing<Bigational>

Figure 2: Overview of polynomial types

structure is enhanced by a table of non-commutative relations, called `RelationTable`, defining the new multiplication. The constructors delegate most things to the corresponding super class constructors and additionally have a parameter for the `RelationTable` to be used. Also the methods delegate the work to the respective super class methods where possible and then handle the non-commutative multiplication relations separately.

The construction of solvable polynomial objects follows directly that of polynomial objects. The type is created by binding the type parameter `C` to the desired coefficient type, e.g. `BigRational`. So we have the type `GenSolvablePolynomial<BigRational>`. Solvable polynomial objects are then created via the respective solvable polynomial factory of type

```
GenSolvablePolynomialRing<BigRational>,
```

which is created by binding the generic coefficient type of the generic polynomial factory to the desired coefficient type, e.g. `BigRational`. A solvable polynomial factory object is created as usual from a coefficient factory object, the number of variables in the polynomial ring and a table containing the defining non-commutative relations with the new operator via one of its constructors. Given an object `coFac` of type `BigRational` as before, a polynomial factory object `spfac` of the above described type could be created by `new`

```
GenSolvablePolynomialRing<BigRational>(coFac,5).
```

This defines a solvable polynomial ring with 5 variables over the rational numbers with no commutator relations. A solvable polynomial object of the above described type can then be created by any method defined in RingFactory, e.g. by `spfac.getONE()`, `spfac.fromInteger(1)`, `spfac.parse( "1" )` or `spfac.random(3)`. Some care is needed to create `RelationTable` objects since its constructor requires the solvable polynomial ring which is under construction as parameter (see section 3.3).

3. FUNCTIONALITY OF MAIN CLASSES

In this section we present the methods defined by the interfaces and classes from the proceeding sections. An overview is given in figure 3.

3.1 Ring elements

The `RingElem` interface (with type parameter `C`) defines the usual methods required for ring arithmetic such as `C sum(C S)`, `C subtract(C S)`, `C negate()`, `C abs()`, `C multiply(C s)`, `C divide(C s)`, `C remainder(C s)`. `C inverse()`. Although the actual ring may not have inverses for every element or some division algorithm we have included these methods in the definition. In a case where there is no such function, the implementation may deliberately throw a RuntimeException or choose some other meaningful element to return. The method `isUnit()` can be used to check if an element is invertible.

Besides the arithmetic methods there are the following testing methods `boolean isZERO()`, `isONE()`, `isUnit()` and `int signum()`. The first three test if the element is 0, 1 or a unit in the respective ring. The `signum()` method defines the sign of the element (in case of an ordered ring). It is also employed in `toString()` to determine which sign to 'print'. The methods `equals(Object b)`, `int hashCode()` and `int compareTo(C b)` are required to keep Java's object machinery working in our sense. They are used when an element is put into a Java collection class, e.g. `Set`, `Map` or `SortedMap`. The last method `C clone()` can be used to obtain a copy of the actual element. As creational method one should better use the method `C copy(C a)` from the ring factory, but in Java it is more convenient to use the `clone()` method.

As mentioned before, the creational aspects of rings are separated into a ring factory. A ring factory is intended to store all context information known or required for a specific ring. Every ring element should also know its ring factory, so all constructors of ring element implementations require a parameter for the corresponding ring factory. Unfortunately constructors and their signature can not be specified in a Java interface. The `RingFactory` interface also has a generic type parameter `C` which is constrained to a type with the ring element functionality (see figure 3). The defined methods are `C getZERO()`, `C getONE()`, which create 0 and 1 of the ring. The creation of the 1 is most difficult, since for a polynomial it implies the creation of the 1 from the coefficient ring, i.e. we need a factory for coefficients at this point. Then there are methods to embed a natural number into the ring and create the corresponding ring element `C fromInteger(long a)` and `C fromInteger(java.math.BigInteger a)`. The others are `C random (int n)`, `C copy(C c)`, `C parse (String s)`, and `C parse (Reader r)`. The `copy()` method was intended as the main means to obtain a copy of a ring element, but it is now seldom used in our implementation. Instead the `clone()` method is used from the ring element interface. The `random(int n)` method creates a random element of the respective ring. The parameter n specifies an appropriate maximal size for the created element. In case of coefficients it usually means the maximal bit-length of the element, in case of polynomials it influences the coefficient size and the degrees. For polynomials there are `random()` methods with more parameters. The two methods `parse(String s)` and `parse(Reader r)` create a ring element from some external string representation. For coefficients this is mostly implemented directly and for polynomials the class `GenPolynomialTokenizer` is employed internally. In the current implementation the external representation of coefficients may never contain white space and must always start with a digit. In the future the ring factory will be enhanced by methods that test if the ring is commutative, associative or has some other important property or the value of a property, e.g. is an euclidian ring, is a field, an integral domain, a unique factorization domain, its characteristic or if it is Noetherian.

3.2 Polynomials

The `GenPolynomialRing` class has a generic type parameter `C` as already explained (see figure 3). Further the class implements a `RingFactory` over `GenPolynomial<C>` so that it can be used as coefficient factory of a different polynomial ring. The constructors require at least a factory for the coefficients as first parameter of type `RingFactory<C>` and the number of variables in the second parameter. A third parameter can optionally specify a `TermOrder` and a fourth parameter can specify the names for the variables of the polynomial ring. Via `TermOrder` objects the required comparators for the `SortedMap` are produced. Besides the methods required by the `RingFactory` interface there are additional

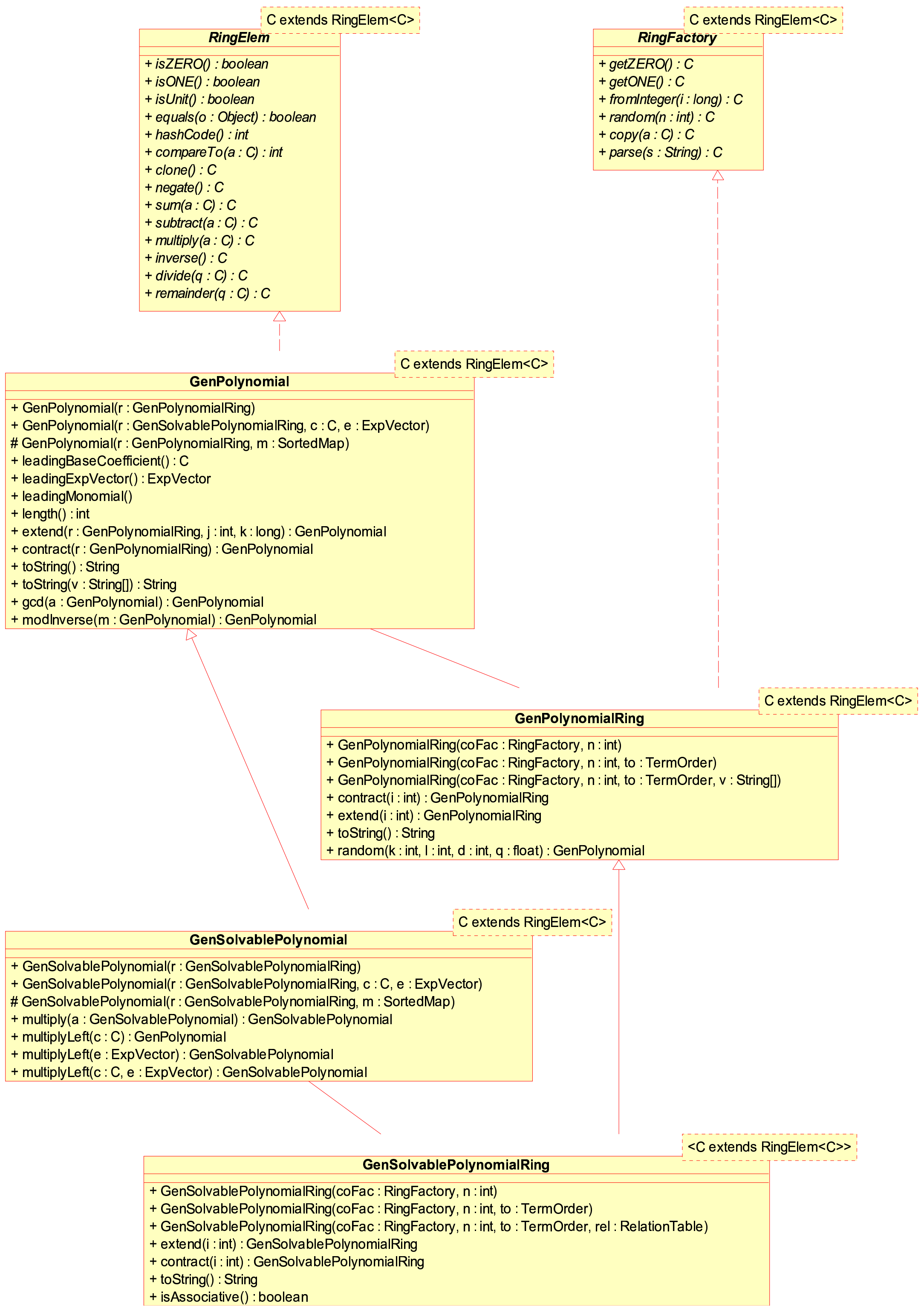

C extends RingElem<C>
RingElem
+ isZERO() : boolean
+ isONE() : boolean
+ isUnit() : boolean
+ equals(o : Object) : boolean
+ hashCode() : int
+ compareTo(a : C) : int
+ clone() : C
+ negate() : C
+ sum(a : C) : C
+ subtract(a : C) : C
+ multiply(a : C) : C
+ inverse() : C
+ divide(q : C) : C
+ remainder(q : C) : C
C extends RingElem<C>
RingFactory
+ getZERO() : C
+ getONE() : C
+ fromInteger(i : long) : C
+ random(n : int) : C
+ copy(a : C) : C
+ parse(s : String) : C
C extends RingElem<C>
GenPolynomial
+ GenPolynomial(r : GenPolynomialRing)
+ GenPolynomial(r : GenSolvablePolynomialRing, c : C, e : ExpVector)
# GenPolynomial(r : GenPolynomialRing, m : SortedMap)
+ leadingBaseCoefficient() : C
+ leadingExpVector() : ExpVector
+ leadingMonomial()
+ length() : int
+ extend(r : GenPolynomialRing, j : int, k : long) : GenPolynomial
+ contract(r : GenPolynomialRing) : GenPolynomial
+ toString() : String
+ toString(v : String[]) : String
+ gcd(a : GenPolynomial) : GenPolynomial
+ modInverse(m : GenPolynomial) : GenPolynomial
C extends RingElem<C>
GenPolynomialRing
+ GenPolynomialRing(coFac : RingFactory, n : int)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder, v : String[])
+ contract(i : int) : GenPolynomialRing
+ extend(i : int) : GenPolynomialRing
+ toString() : String
+ random(k : int, l : int, d : int, q : float) : GenPolynomial
C extends RingElem<C>
GenSolvablePolynomial
+ GenSolvablePolynomial(r : GenSolvablePolynomialRing)
+ GenSolvablePolynomial(r : GenSolvablePolynomialRing, c : C, e : ExpVector)
# GenSolvablePolynomial(r : GenSolvablePolynomialRing, m : SortedMap)
+ multiply(a : GenSolvablePolynomial) : GenSolvablePolynomial
+ multiplyLeft(c : C) : GenPolynomial
+ multiplyLeft(e : ExpVector) : GenSolvablePolynomial
+ multiplyLeft(c : C, e : ExpVector) : GenSolvablePolynomial
<C extends RingElem<C>>
GenSolvablePolynomialRing
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int)
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int, to : TermOrder)
+ GenSolvablePolynomialRing(coFac : RingFactory, n : int, to : TermOrder, rel : RelationTable)
+ extend(i : int) : GenSolvablePolynomialRing
+ contract(i : int) : GenSolvablePolynomialRing
+ toString() : String
+ isAssociative() : boolean
For better readability not all type parameters C are shown.

Figure 3: Overview of class functionality

In this example we show some computations with the polynomial $3x_1^2x_3^4 + 7x_2^5 - 61$ from the introduction.

```
BigInteger z = new BigInteger();
TermOrder to = new TermOrder();
String[] vars = new String[] { "x1", "x2", "x3" };
GenPolynomialRing<BigInteger> ring
 = new GenPolynomialRing<BigInteger>(z,3,to,vars);
GenPolynomial<BigInteger> pol
 = ring.parse( "3 x1^2 x3^4 + 7 x2^5 - 61" );
```

With `toString()` or `toString( ring.getVars() )` the following output is produced. `IGRLEX` is a name for the default term order.

```
ring = BigInteger(x1, x2, x3) IGRLEX
pol = GenPolynomial[
      3 (4,0,2), 7 (0,5,0), -61 (0,0,0) ]
pol = 3 x1^2 * x3^4 + 7 x2^5 - 61
```

Subtraction and multiplication of polynomials is e.g.

```
p1 = pol.subtract(pol);
p2 = pol.multiply(pol);
```

with the following output.

```
p1 = GenPolynomial[  ]
p1 = 0
p2 =  9 x1^4 * x3^8 + 42 x1^2 * x2^5 * x3^4
     + 49 x2^10
     - 366 x1^2 * x3^4 - 854 x2^5 + 3721
```

Figure 4: Example from the introduction

`random()` methods which provide more control over the creation of random polynomials. They have the following parameters: the bit-size of random coefficients to be used in the `random()` method of the coefficient factory, the number of terms (i.e. the length of the polynomial), the maximal degree in each variable and the density of non-zero exponents, i.e. the ratio of non-zero to zero exponents. The `toString()` method creates a string representation of the polynomial ring consisting of the coefficient factory string representation, the tuple of variable names and the string representation of the term order. The `extend()` and `contract()` methods create 'bigger' respectively 'smaller' polynomial rings. Both methods take a parameter of how many variables are to be added or removed form the actual polynomial ring. `extend()` will setup an elimination term order consisting of two times the actual term order when ever possible.

The `GenPolynomial` class has a generic type parameter `C` as explained above (see figure 3). Further the class implements a `RingElem` over itself `RingElem<GenPolynomial<C>>` so that it can be used for the coefficients of an other polynomial ring. The functionality of the ring element methods has already been explained in the previous section. There are two public and one protected constructors, each requires at least a ring factory parameter `GenPolynomialRing<C> r`. The first creates a zero polynomial `GenPolynomial(r)`, the second creates a polynomial of one monomial with given coefficient and exponent tuple `GenPolynomial(r, C c, ExpVector e)`, the third is protected for internal use only and creates a polynomial from the internal sorted map of an other polynomial `GenPolynomial(r, SortedMap< ExpVector, C > v)`. There is no heavy weight contructor accepting a `Map< ExpVector, C >` parameter. Further there are methods to access parts of the polynomial like leading term, leading coefficient (still called leading base coefficient from some old tradition) and leading monomial. The `toString()` method creates as usual a string representation of the polynomials consisting of exponent tuples and coefficients. One variant of it takes an array of variable names and creates a string consisting of coefficients and products of powers of variables. See the example from the introduction in figure 4. The method `extend()` is used to embed the polynomial into the 'bigger' polynomial ring specified in the first parameter. The embedded polynomial can also be multiplied by a power of a variable. The `contract()` method returns a `Map` of exponents and coefficients. The coefficients are polynomials belonging to the 'smaller' polynomial ring specified in the first parameter. If the polynomial actually belongs to the smaller polynomial ring the map will contain only one pair, mapping the zero exponent vector to the polynomial with variables removed. A last group of methods computes (extended) greatest common divisors. They work correct for univariate polynomials over a field but not for arbitrary multivariate polynomials. These methods will be moved to a new separate class together with a correct implementation for the multivariate case if I find some time.

3.3 Solvable polynomials

The `GenSolvablePolynomial` class also has a generic type parameter `C` as explained above. The class extends the `GenPolynomial` class (see figure 3). It inherits all additive functionality and overwrites the multiplicative functionality with a new non-commutative multiplication method. Unfortunately it cannot implement a `RingElem` over itself

```
RingElem<GenSolvablePolynomial<C>>
```

but can only inherit the implementation of

```
RingElem<GenPolynomial<C>>
```

from its super class. By this limitation a solvable polynomial can still be used as coefficient in another polynomial, but only with the type of its super class. The limitation comes form the erasure of template parameters in `RingElem<...>` to `RingElem` for the code generated. I.e. the generic interfaces become the same after type erasure and it is not allowed to implement the same interface twice. There are two public and one protected constructors as in the super class. Each requires at least a ring factory parameter `GenSolvablePolynomialRing<C> r` which is stored in a variable of type `GenPolynomialRing<C>` shadowing the variable with the same name of the super factory type. Via this mechanism also the super class methods will create solvable polynomials. The rest of the initialization work is delegated to the super class constructor.

The `GenSolvablePolynomialRing` class also has a generic type parameter `C`. It extends `GenPolynomialRing` and overwrites most methods to implement the handling of the `RelationTable`. However it cannot implement a `RingFactory` over `GenSolvablePolynomial<C>` but only a `RingFactory` over `GenPolynomial<C>` by inheritance due to the same reason of type erasure as above. But it can be used as coefficient factory with the type of its super class for a different polynomial ring. One part of the constructors just restate the super class constructors with the actual solvable type. A solvable polynomial ring however must know how to perform the non-commutative multiplication. To this end a data structure with the respective commutator relations is required. It is implemented in the `RelationTable` class. The other part of the constructors additionally takes a parameter of type `RelationTable` to set the initial commutator relation table. Some care is needed to create relation tables and solvable polynomial factories since the relation table requires a solvable polynomial factory as parameter in the constructor. So it is most advisable to create a solvable polynomial factory object with empty relation table and to fill it with commutator relations after the constructor is completed but before the factory will be used. In the above example where `spfac` is a factory for solvable polynomials the relations for a Weyl algebra could be generated as follows

```
WeylRelations<BigRational> wl
   = new WeylRelations<BigRational>(spfac);
wl.generate();
```

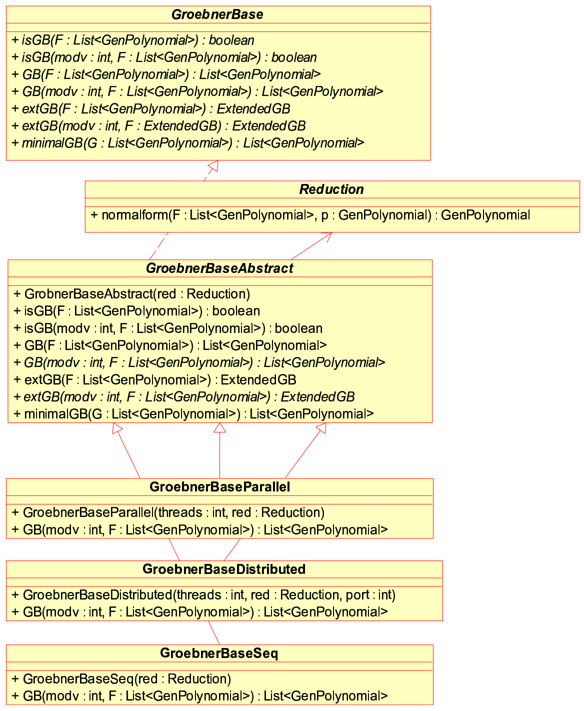
There is also a new method `isAssociative()` which tries to check if the commutator relations indeed define an associative algebra. This method should be extracted to the `RingFactory` interface together with a method `isCommutative()`, since both are of general importance and not always fulfilled in our rings. E.g. `BigQuaternion` is not commutative and so a polynomial ring over these coefficients can not be commutative. The same applies to associativity and the class `BigOctonion`.

4. IMPLEMENTATION

Today the implementation consists of about 100 classes and interfaces plus about 50 JUnit classes with unit tests. Logging is provided by the Apache Log4j package. Moreover there are some Jython classes for a more convenient interactive interface.

Basic data types, such as rational numbers, can directly implement both interfaces `RingElem` and `RingFactory` to avoid the separate implementation of factory classes. More complex data types, such as polynomials implement the interfaces in two different classes. Constructors for basic data types can be implemented in any appropriate way. Constructors for more complex data types with separate factory classes should always require one parameter to be of the respective factory type. This is to avoid the creation of elements with no knowledge of is corresponding ring factory. Constructors which require more preconditions, which are only provided by type (internal) methods should not be declared public. It seems best to declare them as protected.

The implementation of basic arithmetic is based on the `java.math.BigInteger` class, which is itself implemented like GnuMP. Multiplication performance was in 2000 approximately 10 to 15 times faster than that of the respective `SACI` module of MAS [13] (see e.g. the the weblog in [12]). Since we require our big integers to implement the `RingElem` interface, we employ the facade pattern for our `BigInteger` class. Beside this, at the moment the following classes are implemented `BigRational`, `ModInteger`, `BigComplex`, `BigQuaternion` and `BigOctonion`. Using (univariate) generic polynomials we provide an `AlgebraicNumber` class, which can be used over `BigRational` or `ModInteger`, i.e. it implements algebraic number rings with zero or finite characteristic.



GroebnerBase
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : ExtendedGB) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>
Reduction
+ normalform(F : List<GenPolynomial>, p : GenPolynomial) : GenPolynomial
GroebnerBaseAbstract
+ GrobnerBaseAbstract(red : Reduction)
+ isGB(F : List<GenPolynomial>) : boolean
+ isGB(modv : int, F : List<GenPolynomial>) : boolean
+ GB(F : List<GenPolynomial>) : List<GenPolynomial>
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
+ extGB(F : List<GenPolynomial>) : ExtendedGB
+ extGB(modv : int, F : List<GenPolynomial>) : ExtendedGB
+ minimalGB(G : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseParallel
+ GroebnerBaseParallel(threads : int, red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseDistributed
+ GroebnerBaseDistributed(threads : int, red : Reduction, port : int)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>
GroebnerBaseSeq
+ GroebnerBaseSeq(red : Reduction)
+ GB(modv : int, F : List<GenPolynomial>) : List<GenPolynomial>

Figure 5: Groebner Base classes

Generic polynomials are implemented as sorted maps from exponent vectors to coefficients. Helper classes are taken from the Java collections framework, i.e. from the package `java.util`. For the implementation of the sorted map the Java class `TreeMap` is taken. An older alternative implementation using `Map`, implemented with `LinkedHashMap`, has been abandoned due to inferior performance. The monoid of terms consists exponent vectors, i.e. the keys of the `Map` are implemented by the class `ExpVector`. There is only one implementation of exponent vectors `ExpVector` as dense Java array of `long`s. Other implementations, e.g. sparse representation or bigger numbers or `int`s are not considered at the moment. The comparators for `SortedMap<ExpVector,C>` are created from a `TermOrder` class, e.g. by method `getDescendComparator()`. `TermOrder` provides `Comparator`s for most term orders used in practice: lexicographical, graded and term orders defined by weight matrices. The polynomial objects are intended to be immutable. I.e. the object variables are declared `final` and the map is never modified once it is created. One could also wrap it with `unmodifiableSortedMap()` if desired. This design avoids further synchronization on polynomial methods in parallel algorithms.

As explained above non-commutative polynomials defined with respect to certain commutator relations are extended from `GenPolynomial` respectively `GenPolynomialRing`. The commutator relations are stored in `RelationTable` objects, which are intended to be internal to the `GenSolvablePolynomialRing` since they contain polynomials generated from this factory. The `RelationTable` is optimized for a fast detection of commutative multiplication, i.e. relations of the form $x_j * x_i = x_i x_j$ for some i, j. The overhead of computing commutative polynomials with `GenSolvablePolynomial` objects is approximately 20%. The relation table is eventually modified in synchronized methods if new relations be-

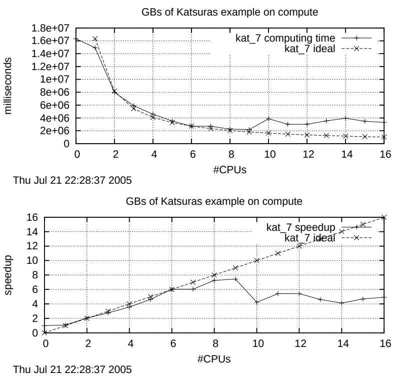


Figure 6: Katsura 7 parallel Groebner base, 16 CPU JDK 1.4, 32 bit JVM, option `UseParallelGC`, Intel XEON 2.7 GHz, 1GB JVM memory, 0 CPUs means sequential

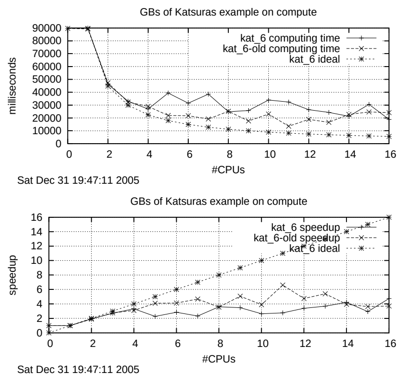


Figure 8: Katsura 6 parallel Groebner base, 16 CPU JDK 1.4, 32 bit JVM, option `UseParallelGC`, Intel XEON 2.7 GHz, 1GB JVM memory, 0 CPUs means sequential

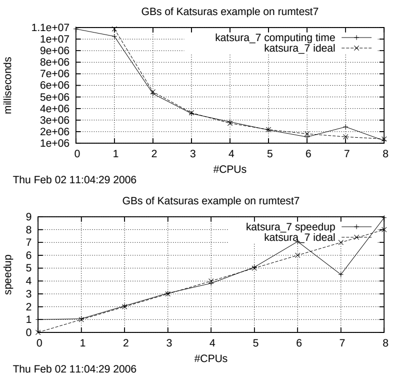


Figure 7: Katsura 7 parallel Groebner base, 8 CPU JDK 1.5, 64 bit JVM, AMD Opteron 2.2 GHz, 1GB JVM memory, 0 CPUs means sequential version

tween powers of variables are computed, e.g. $x_j^{e_j} * x_i^{e_i} = c_{i'j'} x_i^{e_i} x_j^{e_j} + p_{i'j'}$ for some i, j. These new relations are then used to speedup future non commutative multiplications. `GenSolvablePolynomial` implements the non commutative multiplication and uses the additive commutative methods from its super class. As mentioned before, casts are required for the super class methods, e.g.

```
(GenSolvablePolynomial<C>) p.sum(q).
```

The respective objects are however correctly build using the methods from the solvable ring factory.

The class design allows solvable polynomial objects to be used in all algorithms where `GenPolynomials` can be used as parameters as long as no distinction between left and right multiplication is required.

4.1 Groebner bases

As an application of the generic polynomials we have implemented some more advanced algorithms. E.g. polynomial reduction (a kind of multivariate polynomial division algorithm) or Buchbergers algorithm to compute Groebner bases (a kind of a Gaussian elimination for multivariate polynomials). The algorithms are also implemented for solvable polynomial rings (with left, right and two-sided variants) and modules over these rings. These algorithms are implemented following standard object oriented patterns (see figure 5). There is an interface, e.g. `GroebnerBase`, which specifies the desirable functionality, like `isGB()`, `GB()` or `extGB()`. Then there is an abstract class, e.g. `Groebner-BaseAbstr`, which implements as much methods as possible. It further defines the desirable constructor parameters, e.g. a `Reduction` parameter which sets a polynomial reduction engine with suitable properties. Finally there are concrete classes which extend the abstract class and implement different algorithmic details. E.g. `GroebnerBaseSeq` implements a sequential, `GroebnerBaseParallel` implements a thread parallel and `GroebnerBaseDistributed` implements a network distributed version of the core Groebner base algorithm. In 2003 we compared the Groebner base algorithm to a similar version and implementation of MAS [13]. For the big Trinks example the Java implementation was 8 times faster.

4.2 Parallelism

During the work on [14] we developed the ideas for design of parallel and distributed implementation of core algorithms. The Groebner base computation is done by a variant of the classical Buchberger algorithm. It maintains a data structure, called pair list, for book keeping of the computations (forming S-polynomials and doing reductions). This data structure is implemented by `CriticalPairList` and `OrderedPairList`. Both have synchronized methods `put()` and `getNext()` respectively `removeNext()` to update the data structure. In this way the pair list is used as work queue in the parallel and distributed implementations. The parallel implementations scales well for up to 8 CPUs, for a well structured problem (figures 6 and 7) and up to 4 CPUs on a smaller problem (figure 8). Since the polynomials are implemented as immutable classes no further synchronization is required for the polynomial methods. The distributed implementation makes further use of a distributed list (implemented via a distributed hash table DHT) for the communication of the reduction bases and a distributed thread pool for running the reduction engines in different computers. Java object serialization is used to encode polynomials for network transport. Polynomials are only transfered once to or from a computing node, critical pairs are only transfered using indexes of polynomials in the DHT.

5. CONCLUSIONS

We have provided a sound object oriented design and implementation of a library for algebraic computations in Java. For the first time we have produced a type safe library using generic type parameters. The proposed interfaces and classes are as expressive as the category and domain constructs of Axiom or Aldor, although we have not jet implemented all possible structures. The library provides multivariate polynomials and multiprecision base coefficients which are used for a large collection of Groebner base algorithms. For the first time we have presented an object oriented implementation of non-commutative solvable polynomials and many non-commutative Groebner base algorithms. The library employs various design patterns, e.g. creational patterns (factory and abstract factory) for algebraic element creation. For the main working structures we use the Java collection framework. The parallel and distributed implementation of Groebner base algorithms draws heavily on the Java packages for concurrent programming and internet working. The suitability of the design is exemplified by the successful implementation of a large part of 'additive ideal theory', e.g. different Groebner base and syzygy algorithms. With the Jython wrapper the library can also be used interactively.

We hope that the problems with type erasure in generic interfaces could be solved in some future version of the Java language. It would also be helpful if there was some way to impose restrictions on constructors in interface definitions.

In the future we will implement more of 'multiplicative ideal theory', i.e. multivariate polynomial greatest common divisors and factorization.

Acknowledgments

I thank Thomas Becker (who is working or thinking about a C++ implementation of a polynomial template library) for discussions on the subject. Further I thank Aki Yoshida from whom I learned much about object oriented programming and design. Thanks also to the anonymous referees for the suggestions to improve the paper.

6. REFERENCES

[1] M. Y. Becker. *Symbolic Integration in Java.* PhD thesis, Trinity College, University of Cambridge, 2001.

[2] T. Becker and V. Weispfenning. *Gröbner Bases - A Computational Approach to Commutative Algebra.* Springer, Graduate Texts in Mathematics, 1993.

[3] L. Bernardin, B. Char, and E. Kaltofen. Symbolic computation in Java: an appraisement. In S. Dooley, editor, *Proc. ISSAC 1999*, pages 237–244. ACM Press, 1999.

[4] M. Bronstein. Sigma$^{\mathrm{it}}$ - a strongly-typed embeddable computer algebra library. In J. Calmet and C. Limongelli, editors, *Proc. DISCO 1996*, volume 1128 of *Lecture Notes in Computer Science*, pages 22–33. Springer, 1996.

[5] J. Buchmann and T. Pfahler. *LiDIA*, pages 403–408. in Computer Algebra Handbook, Springer, 2003.

[6] M. Conrad. The Java class package com.perisic.ring. Technical report, http://ring.perisic.com/, 2002-2004.

[7] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms.* Springer, Undergraduate Texts in Mathematics, 1992.

[8] J. Grabmaier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook.* Springer, 2003.

[9] G.-M. Greuel, G. Pfister, and H. Schönemann. *Singular - A Computer Algebra System for Polynomial Computations*, pages 445–450. in Computer Algebra Handbook, Springer, 2003.

[10] R. Jenks and R. Sutor, editors. *axiom The Scientific Computation System.* Springer, 1992.

[11] H. Kredel. A systems perspective on A3L. In *Proc. A3L: Algorithmic Algebra and Logic 2005*, pages 141–146. University of Passau, April 2005.

[12] H. Kredel. The Java algebra system. Technical report, http://krum.rz.uni-mannheim.de/jas/, since 2000.

[13] H. Kredel and M. Pesch. *MAS: The Modula-2 Algebra System*, pages 421–428. in Computer Algebra Handbook, Springer, 2003.

[14] H. Kredel and A. Yoshida. *Thread- und Netzwerk-Programmierung mit Java.* dpunkt, 2nd edition, 2002.

[15] V. Niculescu. A design proposal for an object oriented algebraic library. Technical report, Studia Universitatis "Babes-Bolyai", 2003.

[16] V. Niculescu. OOLACA: an object oriented library for abstract and computational algebra. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA Companion*, pages 160–161. ACM, 2004.

[17] A. Platzer. The Orbital library. Technical report, University of Karlsruhe, http://www.functologic.com/, 2005.

[18] E. Poll and S. Thomson. The type system of Aldor. Technical report, Computing Science Institute Nijmegen, 1999.

[19] P. S. Santas. A type system for computer algebra. *J. Symb. Comput.*, 19(1-3):79–109, 1995.

[20] S. Watt. *Aldor*, pages 265–270. in Computer Algebra Handbook, Springer, 2003.

[21] C. Whelan, A. Duffy, A. Burnett, and T. Dowling. A Java API for polynomial arithmetic. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 139–144, New York, NY, USA, 2003. Computer Science Press, Inc.

[22] R. Zippel. Weyl computer algebra substrate. In *Proc. DISCO '93*, pages 303–318. Springer-Verlag Lecture Notes in Computer Science 722, 2001.

Components: A valuable investment for Financial Engineering

Why derivative contracts should be Active Documents

Markus Reitz*
University of Kaiserslautern
Software Technology Group
P.O. Box 3049
67653 Kaiserslautern, Germany
reitz@informatik.uni-kl.de

Ulrich Nögel*
Fraunhofer ITWM
Department of Financial Mathematics
Fraunhoferplatz 1
67663 Kaiserslautern, Germany
noegel@itwm.fhg.de

ABSTRACT

Although component-oriented thinking is quite common to software developers, the paradigm's impact beyond its "native" domain is limited. *Financial Engineering*, a fast-growing discipline that combines finance, applied mathematics and computer science, often uses inflexible straightforward implementations for the underlying mathematical descriptions and models. Missing the benefits of modern software technology, even small variations in the financial products portfolio usually induce huge reimplementation efforts. Instead of concentrating on the creative aspects of contract development, financial engineers have to struggle with hard-to-modify implementations that decrease overall productivity. By providing concepts and techniques that improve and optimise the design and valuation methodology for derivative contracts, ComDeCo[1] transfers the principle of *thinking in components* to this discipline. Using an explorative composition style, problems caused by nearly unbounded flexibility, decreasing time to market periods and shortening product life cycles are tackled effectively. This paper introduces Active Documents as the theoretical background constituting ComDeCo's conceptual foundation. Reasons for the decision to base ComDeCo's domain-specific framework on them and the resulting advantages are discussed. The current state of ComDeCo's Java-based implementation is presented and potential next steps towards the goal of component-oriented financial engineering are sketched. Finally, possible future directions and novel scenarios of application for ComDeCo's results are illustrated.

*Supported by the cluster of excellence *Dependable Adaptive Systems and Mathematical Modeling* (DASMOD) of Rhineland-Palatinate, Germany.
[1]Composable Derivative Contracts, a subproject of DASMOD (`http://www.dasmod.de`).

Keywords

Active Documents, Financial Engineering, Derivative Contracts, Component-Orientation, Java, XML, Web 2.0

1. INTRODUCTION

Albeit being proposed by McIllroy [1] about 40 years ago, the principle of component-oriented software design is still in its early stages. Component markets as envisioned by Cox [2] do not exist (yet), but current software projects are often based on component-oriented frameworks, especially fostering the reuse aspect of iteratively evolving component repositories. Unfortunately, the primary audience of state of the practice technologies is limited to experienced developers. However, software engineers can base their software development efforts on a firm footing of well-tested building blocks by using component-orientation. End-user oriented techniques are rare and often only provide rudimentary functionality.

Transferring component-oriented principles to the end-user domain is not as easy as it may seem at first glance. When targeting end-users who do not possess in-depth technical expertise, composition environments have to be simple and intuitive. The necessity to write glue code as a last resort in case of composition mismatches has to be avoided under all circumstances, because end-users can not be expected to be experienced software developers.

Active Documents, a component-oriented approach which is based on extensions of the well-known metaphor of a document, provide techniques and methodologies that enable *end-user centered component-orientation.* The concept of a conventional document is augmented with the bells and whistles of modern software technology, e.g. visual composition and component-orientation. Moving from physical sheets of paper to computer-based realisations (*Hyperdocuments*) results in tremendous capability extensions: dynamicity replaces inflexible static representations. Nevertheless, users are not overstrained, because fundamental concepts remain the same.

Transitioning from the common application-centric to a task-oriented point of view embodied by Active Documents let application boundaries diminish, eventually disappear completely. End-users are able to customise their working environment according to individual requirements without being forced to follow monolithic upgrade paths which are common nowadays. The reuse aspect being emphasised by current mainstream component technologies like .NET or the Java platform is not of primary interest for end-users. Instead, aspects such as

Adaptability to slight changes and variations of requirements,

Personalisability according to user-specific preferences, usage patterns and habits,

Flexibility to cope with (potentially completely) changing requirements during lifetime,

Openness to handle increasing complexity, changing models of abstraction and usage contexts, and

Simplicity to guarantee hassle-free configuration and tailoring without the necessity of having experts at hand,

are expected to be provided by software. Component-orientation offers the necessary foundation, but to support *component-orientation beyond reuse*, the existing developer-centric approaches have to be augmented by additional concepts and techniques that pave the way for end-user friendly software systems.

2. RELATED WORK

Active Documents represent an evolutionary refinement of *compound document* concepts which initially appeared in the context of Microsoft's *Object Linking and Embedding* (OLE) technology. Enhancements of OLE led to the developer-centric COM [3] component model which is of great importance today. Apple's OpenDoc [4] was an unsuccessful attempt to establish a multi-platform compound document framework. By dropping support in Mac OS X, OpenDoc's practical relevance has diminished completely. Nevertheless, compound document techniques did not totally disappear. Platform specific support with varying degrees of functionality is typically available. GNOME's Bonobo is one of the few approaches that aims at providing an at least cross UNIX foundation for compound documents.

Minerva [5] introduced the notion of an Active Document in context of e-Learning applications, but did not provide a general-purpose framework. In contrast to compound document technologies which solely focus on *geometrical constraints*, a component of an Active Document is packed with structural and semantical constraints that restrict composition, reducing the probability of mismatches.

3. FINANCIAL ENGINEERING

Trading of derivative contracts forms a higher-than-average growing segment of local as well as world-wide financial markets. Financial engineers design and manage derivative contracts whose complexity arbitrarily varies between simple *put options* represented by the *payoff formula*

$$P(T) = \max(K - S(T), 0)$$

and complicated options, e.g. *cliquets* having a set of different upper and lower bounds

$$P(T) = \left(\sum_{i=1}^{N} \max\left(\min\left(\frac{S(t_i) - S(t_{i-1})}{S(t_i)}, F_i\right), C_i\right)\right)^{+}$$

Simply speaking, a financial contract is an agreement between a vendor and a vendee to buy and sell certain "goods" at specified points in time for a certain amount of monetary units. In fact, traded goods do not have to be physical, e.g. a vendor and a vendee could agree that the vendor receives 1000 € for a contract which gives its holder, i.e. the vendee, the right to receive 1500 € when temperature in August '06 falls below the freezing point.

In general, a derivative contract's value somehow depends on the performance of its underlyings, i.e. its value is *derived* from other values. Due to this tremendous degree of flexibility[2], a vendor's creativity concerning contract design is (almost) unlimited. However, derivatives like the aforementioned contract are quite exotic. A typical derivative contract is usually based on the performance of e.g. stocks, bonds, or resources like e.g. oil, gold, or coffee beans.

Example 1. *A vendee expects an improvement of the dollar-euro parity from the dollar perspective within the next six months. He wants to take advantage of this market trend, but in case of false estimation losses should be limited, i.e. the contract should offer some kind of loss protection mechanism.*

In order to trade, i.e. buy or sell, this and any other derivative contract, the fair price has to be determined using appropriate algorithms.

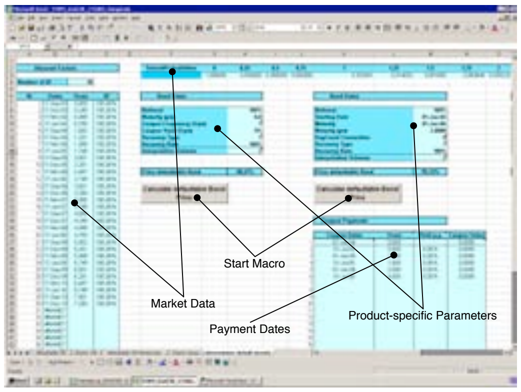
3.1 Excel based spreadsheet prototypes

Rapid prototyping based on Microsoft Excel is a typical approach in financial engineering or quantitative research at financial institutions (see Figure 1). As any other kind of software, prototypes evolve. The initial solution based on pure spreadsheets is extended by *Visual Basic for Applications* (VBA) macros when the need for more sophisticated calculations and applications arises. Increasing performance demands because of numerical burdensome calculations (e.g. Monte Carlo simulations) or intellectual property protection aspects lead to external *dynamic link libraries* (DLLs) whose provided functions are called from the spreadsheet skeleton (see e.g. [6])[3]. A typical prototype in financial engineering uses a three-tiered architecture:

1. An Excel sheet providing the graphical frontend and the platform for rich client applications.
2. VBA code fragments acting as *glue code* extracting and adapting data between the sheet and its lower level layer.
3. A problem specific layer that provides the necessary implementations, subsumed in one or more DLLs.

In practice, separation into completely independent layers is uncommon. Intertwinement dominates, creating high cohesion and strong coupling.

[2]"With derivatives you can have almost any payoff pattern you want. If you can draw it on paper, or describe it in words, someone can design a derivative that gives you that payoff." (Fischer Black, 1995)
[3]There exist more sophisticated techniques to extend Excel, e.g. by using XLL, COM or the .NET framework. However, the general problems in conjunction with the associated drawbacks remain the same.



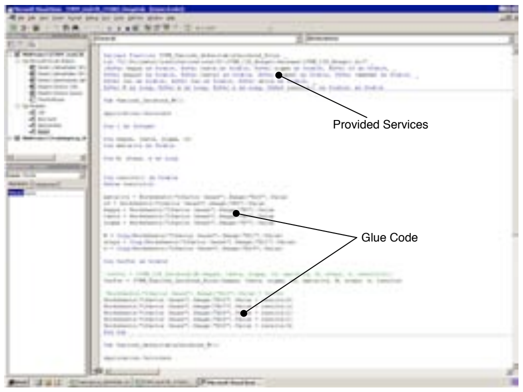


Figure 1: An Excel-based pricing sheet for defaultable *zero coupon bonds* (ZCB) and *credit default swaps* (CDS). The sheet's embedded buttons trigger VBA code execution.

3.2 Prototypes as final products

The pressure of short release cycles hand in hand with monetary constraints favour the acceptance of prototypes instead of mature implementations. What first started as a prototype often becomes an everyday solution[4]. Prototype-inherent problems continue to exist for the whole life cycle.

Inflexibility reduces productivity – Prototypes are almost always inflexible, hard-wired implementations of the underlying mathematical description and models. Missing the benefits of modern software technology, even small variations in the financial products portfolio induce huge reimplementation efforts. Instead of concentrating on the creative aspects of contract design and supporting trading & sales, financial engineers have to struggle with hard-to-modify implementations that decrease overall productivity.

Maintainability, extensions and debugging – Because of high cohesion and strong coupling caused by interwoven layers, the maintenance effort is huge. Local modifications trigger changes in the sheet, the glue code and in the underlying DLLs. Proprietary APIs and complex architectures cause time consuming debugging cycles.

Archaic Design – In the past, spreadsheet solutions were tied to plain C/C++ DLL functions. Besides design issues, a large amount of time was spent on low level programming aspects like pointers, arrays, and garbage collection. While the underlying C++ code may obey the concepts of object-orientation, most implementations stayed close to an inflexible procedural design because of limitations in the wrapper code. Even today, design patterns (see e.g. [8]), modular design, and component-orientation are still uncommon.

Captured in proprietary solutions – Even without using COM or .NET, typical spreadsheet solutions are intrinsically tied to the Excel application platform. Migration or cross platform usage is at least limited and often illusory. Although many financial institutions are using the Windows platform, there is a tendency towards open systems like Linux in combination with OpenOffice. The importance of cross platform solutions will grow in the future, making a closed-source ecosystem more and more uninviting.

[4]Exceptions of this rule exist, e.g. [7].

4. COMDECO

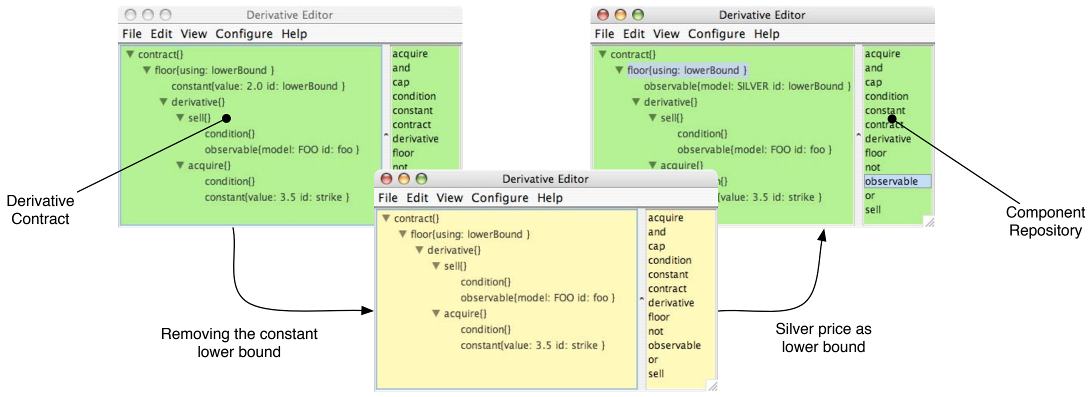
Decreasing time to market cycles and the permanent demand for all new products increases the pressure on financial engineers. To assure a company's market position, design and valuation of derivative contracts have to be carried out as optimal as possible. A newcomer's market share is enlarged when using improved workflows and already established market players are able to strengthen their share of the market. In fact, financial engineers have to face similar problems as their colleagues in the software development domain.

ComDeCo [9] aims at providing techniques and concepts which offer a high degree of scalability, so the increasing complexity of contract design and valuation can be tackled efficiently. Albeit not being a silver bullet for all kinds of problems in this domain, component-orientation is able to provide the foundation for optimised design and valuation frameworks. A derivative contract is modeled as an Active Document which is manipulable by using visual composition gestures, e.g. drag and drop. The domain-specific framework of ComDeCo provides *end-user compatible composition facilities* tailored to the demands of financial engineering and an adaptable valuation architecture that is able to price almost arbitrary contracts. In ComDeCo's context, the term "active"[5] refers to

- the ability to check and enforce consistency constraints for derivative contracts, i.e. avoid meaningless design attempts.
- the ability to automatically perform context-specific adaptations, e.g. selection of the best-fitting valuation algorithm and model.

The possibility to describe derivative contracts using functional programming languages [10] is a hint indicating the

[5]However, in the context of the generic general-purpose Active Document framework *Omnia*, dynamicity is discussed in a more abstract way based on the component and composition model.


Derivative Editor
File Edit View Configure Help
Derivative Contract
Component Repository
Removing the constant lower bound
Silver price as lower bound

```
▼ contract{}
  ▼ floor{using: lowerBound }
      constant{value: 2.0 id: lowerBound }
    ▼ derivative{}
      ▼ sell{}
          condition{}
          observable{model: FOO id: foo }
      ▼ acquire{}
          condition{}
          constant{value: 3.5 id: strike }
```

```
acquire
and
cap
condition
constant
contract
derivative
floor
not
```

```
▼ contract{}
  ▼ floor{using: lowerBound }
    ▼ derivative{}
      ▼ sell{}
          condition{}
          observable{model: FOO id: foo }
      ▼ acquire{}
          condition{}
          constant{value: 3.5 id: strike }
```

```
acquire
and
cap
condition
constant
contract
derivative
floor
not
observable
or
sell
```

```
▼ contract{}
  ▼ floor{using: lowerBound }
      observable{model: SILVER id: lowerBound }
    ▼ derivative{}
      ▼ sell{}
          condition{}
          observable{model: FOO id: foo }
      ▼ acquire{}
         ){}
          {value: 3.5 id: strike }
```

```
acquire
and
cap
condition
constant
contract
derivative
floor
not
observable
or
sell
```

Figure 2: Starting from a simple contract already available in the contract repository, the constant lower bound is substituted by a protection whose value is the silver market price. Consistent states are indicated by a green background, whereas transitional states are signaled by a yellow background color.

applicability of hierarchical description techniques. ComDeCo uses a domain-specific adaptation that models derivative contracts as a special variant: *Hierarchical* Active Documents (see Figure 2). Being hierarchical, a XML based representation is easily derived (see Figure 3). Besides being an alternative to a pure visual composition process, a XML-based representation allows for further manipulation operations by third parties (e.g. *XSL* or *XML Schema* tools) and is an excellent foundation for an open, i.e. non-proprietary, design workflow.

4.1 Building blocks

ComDeCo provides a set of typical *financial components* that can be used to create almost arbitrary contracts. The *derivative* component is one of those core entities. Having a *sell* and an *acquire* component as its children, the simplest of all derivative contracts is represented by a *virtual sale* and a *virtual purchase*, whose execution conditions are represented by appropriate components. By performing composition operations that are based on the *decorator pattern*, the user has the choice to either augment already available contracts with desired properties or start from scratch. The initial contract in Figure 3 consists of a *derivative* component decorated with an upper bound performance participation limit property represented by the encapsulating *cap* component.

The flexibility of an Active Document system stems from its variable component repository, which is filled according to specific user requirements. Adding components to the repository makes all other parts of the system, e.g. composition and valuation facilities, automatically aware of the supplemental functionality.

4.2 Separation of concerns

In case of fair price calculations, there is at least one additional Active Document that is necessary to perform proper contract valuation: the *market specification. Observable* components of the derivative contract express dependencies to other (potentially external) components refered to by their *model* parameter. These hooks[6] are bound by partially merging the derivative contract and its corresponding market specification (see Figure 3). Separation of these two aspects introduces an additional degree of freedom: Having the opportunity to choose from a set of market specifications, the financial engineer is able to run through several case scenarios just by combining appropriate documents. After having designed an individual contract (see Section 6.1), a potential customer is able to get an impression of a derivative's performance in different scenarios, e.g. bullish or bearish markets.

4.3 Component-oriented pricing engine

When determining the fair price of a given derivative contract, available common mathematical approaches may be categorised into three groups

Closed-form solutions require limited computational resources, but the set of available formulae is restricted and can by no means keep pace with the permanently growing plethora of derivative contracts.

Monte Carlo simulation provides an almost general-purpose, but computationally intensive solution. Unfortunately, pricing *american style* derivatives is not straightforward.

Tree-based algorithms are able to price *european* as well as *american style* derivatives and usually show up fast convergence. However, the memory footprint may impose limitations on usage in multi-dimensional settings.

As the ComDeCo framework allows for the construction of arbitrary contracts, a categorisation scheme supporting the determination of the best-fitting approach has to be developed. None of the above-mentioned standard approaches is a panacea, as applicability depends on certain properties of the derivative contract to be valuated. For that reason, the pricing engine has to obey a strategy that takes closed-form solutions into account whenever possible, uses tree-based algorithms as the default approach and offers Monte Carlo simulation on demand.

Although being a de facto market standard, the Black & Scholes model [12] reaches its limits in case of complex derivative contracts. To cope with the flexibility imposed

[6]The concept of a hook resembles similarities to the notions and techniques of [11].

```
<contract>
  <cap using="upperBound">
    <constant id="upperBound" value="20.0"/>
    <derivative>
      <sell>
        <condition/>
        <observable id="foo" model="FOO"/>
      </sell>
      <acquire>
        <condition/>
        <constant id="strike" value="3.5"/>
      </acquire>
    </derivative>
  </floor>
</contract>
```

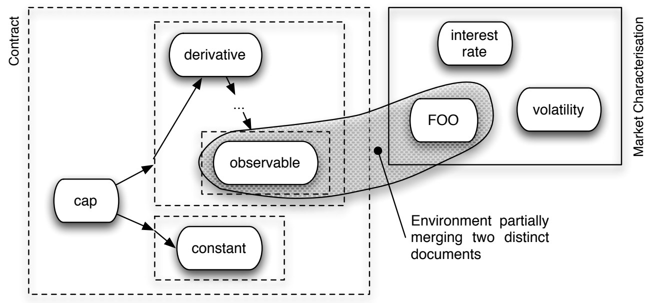


Figure 3: The XML representation of a derivative contract modeled by a hierarchical Active Document. Boxes and geometric objects of grey colour represent environments. Using *local messages only* environments, the sketched message propagation scheme is enforced by the runtime system.

by the composition frontend, additional approaches have to be taken into account, e.g. *local* and *stochastic volatility models*.

The pricing engine has to solve an assigment problem which associates each given contract with the best-fitting algorithm whereas the decision is based on an appropriate categorisation scheme. Additionally, a repository of valuation models has to be managed, allowing for both: automatic model selection based on the categorisation results and manual selection triggered by user interaction.

Component-orientation guarantees the required flexibility. Valuation algorithms as well as models are realised as *plugins*, a light-weight component model which allows for easy extensions of the pricing engine. New algorithms and models may be loaded during system startup without the need to recompile parts or even the whole system. Depending on additional meta-information, the pricing engine is able to detect algorithm-model mismatches, providing error messages or even reject the corresponding valuation attempt at all. The meta-information gained by querying the contract's ACTIVE DOCUMENT representation is used to drive the categorisation mechanism of the pricing engine.

4.4 Backtesting

Besides valuation of designs using the services of the pricing engine, testing based on real world data, so-called *backtesting*, is another method to get an impression of a contract's real world suitability. Making use of historic performance data, a contract may be stepwise executed in a fashion similar to debug sessions common in software development. In the end, COMDECO's backtesting engine (*Retro*) will be able to integrate data from various sources made available via web services or proprietary APIs. Currently, Yahoo! Finance data is useable for rudimentary backtesting purposes.

Aside from this virtual post mortem analysis, data made available by *Retro* may be used for pricing engine calibration. Parameters like the market volatility σ could be calculated from available data instead of being entered manually by the user. Additionally, real-time market data may be used by the valuation engine (see Figure 4).

4.5 Advantages

Financial engineers have used pure mathematical representations for decades, so why should they switch to ACTIVE DOCUMENTS instead?

Visual Composition allows for a simplified yet powerful design process. Instead of fiddling with hard-coded solutions, financial engineers are able to concentrate on the creative part, i.e. design, not bug hunting.

Domain Specific Modeling provides a solution to the impedance mismatch financial engineers are usually confronted with when using general-purpose programming languages for derivative contract design and valuation. COMDECO's components represent a high-level vocabulary financial engineers can make use of when designing contracts. Due to the available supplemental meta-information, the valuation process is significantly simplified.

Constraint Checking decreases the number of potential composition errors by letting only valid composition attempts become effective. For example, accidently mixing an upper bound C and a lower bound F, so F is taken as the upper bound and C as the lower bound results in a broken contract[7]. This and similar defects are automatically detected because of constraints that operate on the component level (e.g. *cap* and *floor*) as opposed to the level of mathematical expressions (e.g. *max* and *min*).

Flexibility allows for system enhancements without the need to reinvent the wheel again and again. Additional building blocks are deployed by adding the components to the repository, making them available for all composition attempts. For example, adding smoothing operators to the set of contract building blocks is just a matter of developing and deploying appropriate *arithmetic mean*, *geometric mean*, and *harmonic mean* components. The composition facilities and the valuation engine are automatically able to handle the additional functionality without any user interaction. Customer-dependent configurations of the component repository are easily created and managed.

[7]The corresponding payoff formula would be something similar to $P = \min(\max(X, F), C)$, which is independent from the value of X in case of $F \geq C$.

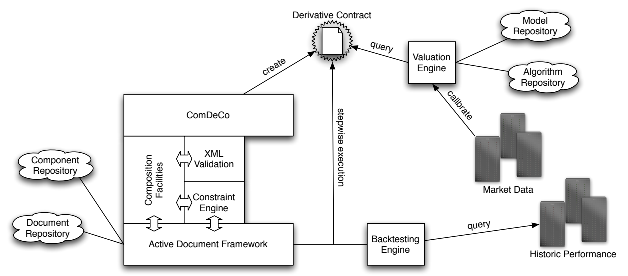


Figure 4: A conceptual overview of the overall system architecture of ComDeCo as discussed in this paper.

5. ACTIVE DOCUMENTS

After having sketched the overall purpose of project ComDeCo, the following sections give a brief overview of general concepts and principles obeyed by the end-user oriented component technology of Active Documents [13]. Technically, ComDeCo uses *Omnia* as its foundation, adding domain-specific features as the need arises. The Java-based *Omnia* system, which is a general-purpose framework for the creation and management of arbitrary types of Active Documents, is developed in parallel during the ComDeCo project. The next sections focus on its general properties[8].

5.1 Component Model

According to the definition brought up during the Workshop on Component-Oriented Programming in 1996

> "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties."

Omnia's component model is based on this foundation, but the rather unspecific definition is further refined, leading to the following component properties:

1. A component subsumes three kinds of information
   (a) Implementation information based on Java-VM bytecode to guarantee cross-platform usage.
   (b) Provided and required services specification using XML-based description schemes.
   (c) Structural and semantical composition constraints which are defined using appropriate XML techniques and declarative rule based specifications.
2. A component may be subject to composition operations only if its structural and semantical composition constraints are satisfied or will be satisfied by further additive composition operations in the future.

[8]Note that the document metaphor as perceived by the end-user is not explicitly enforced by the general concepts and principles. Using the *Omnia* framework, it is possible to support completely different metaphors, too. "Active Documents" that do not even have visual representations are imaginable.

3. The set of services provided by the component is dependent on its neighbourhood[9] and may be subject to changes during runtime.
4. The employed communication model is based on the principle of *partial anonymity* and enforces loose coupling of involved entities.

5.2 Environments

Environments act as containers that are able to embed components and other environments, therefore allowing for arbitrarily nested structures. Besides structuring aspects, environments control message propagation of their embedded entities. Two kinds of messages are distinguished:

Intra-environmental messages are exchanged between entities being embedded in the same environment.

Inter-environmental messages pass environmental borders, i.e. if an intra-environmental message is intercepted by another embedded environment, it becomes an inter-environmental one.

Only inter-environmental messages are manipulable by the controlling environment, e.g. the controlling environment may block message propagation or modify the message before it is multicasted to its embedded entities. Besides general-purpose environments which may be arbitrarily configured according to specific needs, the framework currently provides special variants:

Local Messages Only environments block all but intra-environmental messages. This category of environment is heavily used in case of hierarchical Active Documents with ComDeCo being one example.

Fallback environments react to messages only if there is no other responding entity available. As a consequence, at most one fallback environment is embedded into another environment. This category of environment is useful when modeling inheritance-like message processing behaviour.

Example 2. *GUI frameworks such as SWING provide simple (e.g.* `JButton`*) and complex (e.g.* `JFileChooser`*) widgets. Unfortunately, complex widgets are usually instantiated using hard-wired simple widget constructor calls. If, for instance, a developer enhances the* `JButton` *class, instances of* `JFileChooser` *still use* `JButton` *instead of the enhanced version. The* factory design pattern *provides a solution, but even in this case, source code has to be altered each time a substitution occurs. As Figure 5 illustrates,* Omnia *allows for the realisation of a* dynamic factory mechanism, *making manual source code changes obsolete.*

Directory-based environments are used in case of filesystem-based composition descriptions. Directories represent environments and the embeds-relation is expressed by the directory tree structure. This environment category is especially useful for rapid prototyping purposes. For example, ComDeCo's market specification is currently realised in that way.

[9]The neighbourhood is the set of entities that are able to receive messages from a component and / or send messages to it.

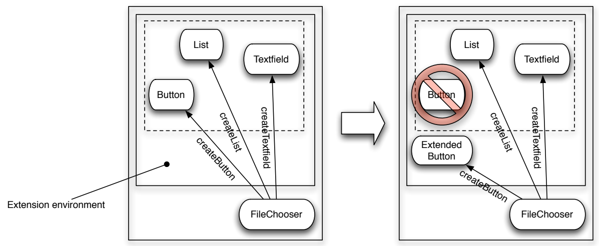


Figure 5: A GUI is extended by putting substitution components into the extension environment. Due to the *fallback environment* surrounding the core building blocks, substitutes dominate native components. Higher level widgets such as a file chooser automatically use the substitutes in case of framework evolutions.

These types of environments may be mixed arbitrarily, providing the building blocks to resemble to the requirements of the specific domain of application.

5.3 Communication Model

Interaction with an ACTIVE DOCUMENT occurs by triggering message sending. Messages are propagated to the entities according to the propagation constraints defined by environments. Communication is based on the following assumptions.

Partial anonymity – A sending entity is unable to determine the set of entities that are able to react to a message. On the other hand, a receiving entity is able to distinguish senders.

Ship and pray – It is possible (and not unusual) that a message being sent is not processed by any entity in the neighbourhood. A sending entity may not make any assumptions about the number of entities responding to a service request.

No Message Monotonicity – The reaction / response pattern showing up as a consequence of a message being sent to the neighbourhood may change over time.

Contextual polymorphism – A component may be embedded into more than one environment. Its reaction to messages depends on the target environment the incoming message was received from.

EXAMPLE 3. *An* ACTIVE DOCUMENT *may be rendered to a static representation, e.g. HTML or PDF by sending an appropriate* render *message. Embedding each component constituting the* ACTIVE DOCUMENT *into several special render environments, e.g. a HTML and a PDF rendering environment, allows for different output just by injecting the* render *message into the proper environment.*

Obeying these rules, *Omnia*'s runtime system is able to guarantee loose coupling of entities constituting an ACTIVE DOCUMENT. From a component's point of view, its neighbourhood is a black box from which messages are received and to which it sends messages. A component is unaware of its neighbours and the overall system structure.

5.3.1 Implementation details

A consequence of these assumptions is the fact that an environment's interface, i.e. the set of messages its embedded entities are able to react to, may change during runtime - it is dynamic[10].

EXAMPLE 4. *Suppose having a set of components providing graphic conversion services transforming an application's internal representation to one of a plethora of formats, e.g. JPEG, PNG, or GIF. Depending on the embedded entities, the corresponding environment is able to provide conversion services by forwarding incoming messages. Adding components to the environment during runtime increases the number of available services, enlarging the environment's interface.*

Although being influenced by the programming language *Objective-C*, Java does not support arbitrary method invocations. Any method being called on an object has to be declared in advance, i.e. statically. Being a moving target, an environment's interface is not easily expressed. By introducing *message objects* which represent messages being exchanged between entities of an ACTIVE DOCUMENT, *Omnia* provides an adequate solution to this problem. Java's reflection capabilities provide the necessary infrastructure to perform on-demand mapping of message objects to appropriate method invocations. Besides solving the dynamic interface problem, making messages first class citizens of the runtime system provides additional advantages, a simplified message filtering and manipulation facility being just one example.

Bytecode inspection. We are currently investigating the capabilities of the ASM framework [14] to extract the component's *required interface* upon loading and instantiation by the runtime system. Although not being able to provide an answer in the general case, bytecode inspection may also be used to detect *defunct components*, i.e. components whose *required interface* can not even be partially satisfied by the neighbourhood it is injected into. Bytecode inspection should be regarded as a supplemental mechanism that is able to check conformance of a component's explicit required interface specification with the actual implementation, avoiding runtime errors upon service invocation.

5.4 Composition

After having introduced the building blocks of any ACTIVE DOCUMENT, namely *components* and *environments*, the following sections discuss the overall composition principles. Generally speaking, the creation of an ACTIVE DOCUMENT adheres to the following steps:

1. Definition of the overall structure by grouping and nesting environments.

2. Deployment of components into environments, creating appropriate neighbourhood relations.

To guarantee only valid compositions, each component provides specifications describing structural and semantical conditions that have to be met during and after composition.

[10]Although there is no constraint that forbids components having changing services, i.e. varying interfaces, it is not the common case.

5.4.1 Structural Constraints

Structural constraints control composition by restricting valid neighbourhood configurations for each component. As stated earlier, a XML-based representation is easily derived in case of hierarchical ACTIVE DOCUMENTS. The *Omnia* framework offers a high level API to express structural constraints, resembling the capabilities of Relax NG [15], XML Schema [16] and Schematron [17], operating on the Java platform's XML APIs in conjunction with third party library usage.

EXAMPLE 5. *The following code excerpt sketches the structural constraint specification for a* floor *component.*

```
...
spec.beginSpecificationFor(spec.getTypeFor(this));
  spec.addEntry(spec.getGroupFor(observable.class));
  spec.beginChoice();
    spec.addEntry(spec.getTypeFor(derivative.class));
    spec.addEntry(spec.getTypeFor(cap.class));
  spec.endSmart();
spec.endTypeSpecification();
...
```

The component's lower bound may be represented by observable-*like components, i.e. components that belong to the same group as the* observable. *Either* derivative *or* cap *components are encapsulated.*

This description is rendered back to a functional equivalent XML Schema[11] or Relax NG representation, possibly complemented by Schematron directives.

EXAMPLE 6. *The code snippet representing the structural constraint specification for a* floor *component is rendered to the following XML Schema fragment.*

```
...
<complexType name="floor">
  <choice>
    <sequence>
      <group maxOccurs="1"
             minOccurs="1"
             ref="ConstantOrObservable"/>
      <choice>
        <element maxOccurs="1"
                 minOccurs="1"
                 name="derivative"
                 type="derivative"/>
        <element maxOccurs="1"
                 minOccurs="1"
                 name="cap"
                 type="cap"/>
      </choice>
    </sequence>
    <sequence>
      <choice>
        <element maxOccurs="1"
                 minOccurs="1"
                 name="derivative"
                 type="derivative"/>
        <element maxOccurs="1"
                 minOccurs="1"
                 name="cap"
                 type="cap"/>
      </choice>
      <group maxOccurs="1"
             minOccurs="1"
             ref="ConstantOrObservable"/>
    </sequence>
  </choice>
  <attribute name="using"
             type="xs:string"
             use="required"/>
</complexType>
...
```

Further types of specifications, e.g. transitional or attribute-reduced ones, may be derived to support certain aspects of the composition process.

[11]Note that due to the *Unique Particle Attribution Rule* of XML Schema, the necessity to render the high level description into several different XML Schema descriptions may occur. Nevertheless, this behaviour is transparent for the user.

Validating structural consistency therefore means validating a hierarchical ACTIVE DOCUMENT's XML representation against schema descriptions generated from specification fragments provided by each component managed by the runtime system. Specification fragments are retrieved during class loading using Java's reflection capabilities, utilising specialised class loader variants.

5.4.2 Semantical Constraints

Semantical constraints control composition behaviour beyond pure structural aspects. For this purpose, a rule engine is used, achieving a clear separation between implementation and logic. Based on a declarative programming style, the component developer is able to specify *what* should be checked without being forced to think about *how* these checks are performed, further improving the loose coupling characteristics of ACTIVE DOCUMENT components.

In principle, two categories of rule engines are potential candidates when implementing the semantic checking layer of an ACTIVE DOCUMENT system:

Backward Chaining engines react to requests which they receive from their clients - they are *demand driven*. The programming language PROLOG is a typical member of this category.

Forward Chaining engines follow an event-based approach, i.e. rule execution is triggered by changes in the engine's entity universe without the need for explicit rule base queries. Engines based on this principle typically use RETE [18] or LEAPS [19] algorithms to perform efficient rule selections and executions.

Omnia uses the Java-based open source engine Drools [20] which implements a variant of RETE (ReteOO) as well as the LEAPS algorithm, therefore following the forward chaining approach. Using the Drools API, an impedance mismatch free integration of rule based programming principles into the Java-based *Omnia* framework is possible. Each entity constituting an ACTIVE DOCUMENT is also represented in the document's working memory managed by the Drools engine, automatically triggering rule execution in case of changes.

5.4.3 End-users' point of view

End-users are unaware of all these technical details happening in the background. For them, a component is just a smart building block which is able to provide the necessary information to the runtime system, making any composition attempt supervisable by the runtime environment. The component model advocated by *Omnia* strictly separates the roles of developers and end-users, so significant programming skills are not required when using components provided by component developers.

Explorative Composition Style. Depending on the constraints that are derived from the contents of the component repository, a given ACTIVE DOCUMENT may be in one of the following states.

Inconsistent States represent all circumstances in which a constraint violation occurs that can not be corrected by adding further components to the document.

Transitional States represent all circumstances with constraint violations that can be corrected additively, i.e. by letting further components join the document.

Consistent States represent all circumstances in which all constraints are satisfied.

In contrast to usually two-valued offline composition approaches, typical in case of developer-centric component models, Active Documents follow the above-mentioned three-valued online composition scheme. ComDeCo's SWING-based derivative editor utilises a colour pattern based on an intuitive traffic light analogon: red colour indicates inconsistent, yellow colour indicates transitional and green colour indicates consistent states (see Figure 2).

6. OUTLOOK & FUTURE WORK

This section sketches possible usage scenarios of Active Documents beyond ComDeCo's (primary) goals. Besides financial engineers as main audience, new opportunities open up due to the end-user orientation of Active Documents. Additionally, further directions the ongoing project ComDeCo may investigate in future working packages are discussed.

6.1 Derivative contracts for the masses

Small and medium-scale financial investors are currently not targeted by OTC[12] products, because of large fixed costs and administrative efforts. Only a large-scale investor's[13] financial and assets position is sufficient to make OTC a viable option. Nevertheless, small-investor's demand for alternatives to traditional investments like bonds and stocks has permanently grown, further accelerated by the new economy crisis. Semi-individual products like *index*, *bonus* or *basket certificates* provide alternatives, but they only partially satisfy customer demands.

With online banking being a mainstream service nowadays, adding a derivative contract construction tool kit to the set of portfolio management features could be a next step. Customers would be able to design derivatives according to individual preferences and market sentiments in a web based online portal. Using the tool chain provided by ComDeCo, the service provider is able to valuate, and depending on the results, accept or reject the customer's contract proposal instantaneously. Upon acceptance, the contract is added to the customer's portfolio in real time followed by an automatic account charge based on the results of the valuation process.

6.2 e-Learning

The Minerva framework already demonstrated general applicability of Active Document concepts in case of the e-Learning domain. Using a light-weight component model, Minerva's components are specifically tied to the domain-specific application framework, therefore lacking properties of a general-purpose Active Document system. Merging the insights of the EU-funded Easycomp project [21] with the experiences gained during ComDeCo will lead to an unification of principles that are incorporated back into the general-purpose *Omnia* framework.

[12]Over The Counter, i.e. financial products specifically tailored according to individual customer needs.

[13]For example, hedgefonds or insurance companies.

6.3 Vanishing application boundaries

Although component-orientation is one of the dominating principles in software development these days, the end-user often does not profit from this paradigm switch. Applications are still monolithic and instead of task-oriented usage patterns, application-centric views are enforced. Rather than fair and flexible upgrade policies, customers are confronted with the choice of "All or nothing" and inter-vendor interoperability is often illusory.

Active Documents provide a transition from application-centric to task-oriented man-machine interaction. Software systems adapt to the user's needs and not vice versa. Personalising the system is done by providing an appropriate set of components. With emerging demands, the component repository is adapted, offering a well-tailored solution for the tasks to be solved. Taking this approach, application boundaries tend to fade away, eventually vanishing completely. As components may be used by any Active Document aware application, there exists no locally bounded area of impact when adding or removing components. Going one step further, i.e. thinking in terms of documents instead of applications, concludes the transformation, letting applications completely disappear as all functionality is spread over the whole system. This point of view shares conceptual similarities with the ideas brought up by Raskin [22].

6.4 Bridging old and new

Termsheet implementations based on Microsoft Excel in conjunction with the necessary additional functionality provided by DLLs represent common practice. Because of their similarity to physical paper sheets, the spreadsheet metaphor is easily understood by users. Solutions based on this principle do not scale up well in case of increasing complexity and the limited composition support often causes reinventions of the wheel. Development expertise is needed to create appropriate spreadsheet(s) according to mathematical formulae developed by a financial engineer. As a personal union of software developer and financial engineer is not a common case, there is almost always a time gap between design and implementation. Despite these disadvantages, it would be starry-eyed to expect an immediate switch over to the principles advocated by ComDeCo. A large amount of knowledge conglomerated in many spreadsheets exists, making legacy support a critical aspect. A legacy bridge based on the OpenOffice suite framework is one possibility. Because Java belongs to OpenOffice's default SDK languages, a (nearly) toll-free bridging between the office legacy layer and the *Omnia* framework is an option.

Besides that, using the OpenOffice framework as an alternative presentation layer for specific variants of Active Documents is a promising way to go, future developments of the *Omnia* framework may concentrate on.

6.5 Web 2.0

Although being in permanent flux, the notion *Web 2.0* subsumes principles and concepts that have the potential to supersede the predominantly static web of these days. AJAX[14]-driven web applications in conjunction with the *semantic web*'s conceptual framework lead to significant improvements. Integrating the Active Document framework

[14]AJAX is an abbreviation for Asynchronous JavaScript And XML, see [23] for more details.

into the conceptual foundation of next generation web technologies is a future research target. To name just two out of a large collection of questions arising in this context:

1. How could principles and practices of Web 2.0 be integrated into the existing ACTIVE DOCUMENT framework, providing an all-embracing conceptual approach?
2. To what extend could Web 2.0 benefit from component-oriented principles in general and from ACTIVE DOCUMENT technology in particular?

ACTIVE DOCUMENT technology might provide extensions for existing web-engineering frameworks with respect to end-user compatible technologies. As a consequence, future web browsers might serve as runtime environments for ACTIVE DOCUMENTS.

7. CONCLUSIONS

ACTIVE DOCUMENTS provide end-user compatible component orientation. Using this technology, adaptation tasks are autonomously performed by the user without the need to have an expert at hand and without being bound to specific applications. Composition is controlled by the runtime system using specification fragments each component is shipped with, resulting in a guided and explorative composition style.

COMDECO uses *Omnia* as its foundation, adding domain-specific functionality as needed. Derivative contracts of increasing complexity are designed by using the document metaphor advocated by ACTIVE DOCUMENTS. Besides a scalable design workflow, meta-data provided by ACTIVE DOCUMENTS supports design and implementation of efficient and automatically adapting valuation and composition facilities.

Among other things, *Omnia*'s component model makes use of Java's reflection capabilities to implement flexible runtime loading mechanisms and a communication scheme which supports loosely coupled component interaction by making messages first class citizens. The Java platform's rich set of APIs and the large pool of actively evolving (open source) libraries provide the necessary ingredients to significantly decrease *Omnia*'s overall development time. For example, the platform's built-in XML support eases development of structural constraint checking facilities and the availability of different rule engines simplifies the implementation of the semantical constraint checking layer. Moreover, language & platform maturity and cross-system availability make Java a good choice for academic as well as industrial applications.

8. REFERENCES

[1] M. D. McIllroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report of a conference sponsored by the NATO Science Committee (Garmisch, Germany - October 7-11, 1968)*. NATO, Scientific Affairs Division, 1969.

[2] B. J. Cox and A. J. Novobilski. *Object-Oriented Programming - An evolutionary approach.* Addison-Wesley Publishing Company Inc., second edition, 1991.

[3] D. Box. *Essential COM.* Addison Wesley Publishing Company Inc., 1999.

[4] Apple Computers Inc. *Inside Macintosh: OpenDoc Programmer's Guide.* Addison Wesley Publishing Company Inc., 1996.

[5] M. Reitz and C. Stenzel. Minerva: A component-based framework for Active Documents. In Aßmann et al., editor, *Proceedings of the Software Composition Workshop (SC 04)*, number 114 in Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[6] S. Dalton. *Excel Add-In Development in C/C++: Applications in Finance.* Wiley, 2005.

[7] Deutsche Bank. Microsoft E2A Programm. *Quantessence*, 2(1):17–37, 2001.

[8] M. Joshi. *C++ Design Patterns and Derivatives Pricing.* Cambridge University Press, 2005.

[9] M. Reitz and U. Nögel. Derivative Contracts as Active Documents - Component-Orientation meets Financial Modeling. In *Proceedings of the 7th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE 06)*, 2006.

[10] S.L. Peyton Jones and J.-M. Eber. *How to write a financial contract*, volume Fun Of Programming of *Cornerstones of Computing.* Palgrave Macmillan, 2005.

[11] U. Aßmann. *Invasive Software Composition.* Springer Verlag, first edition, 2003.

[12] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–659, 1973.

[13] M. Reitz. Active Documents - Taking advantage of component-orientation beyond pure reuse. In *Proceedings of the Workshop on Component-Oriented Programming (WCOP 06)*, 2006.

[14] E. Bruenton, R. Lenglet, and T. Coupaye. ASM - a code manipulation tool to implement adaptable systems. 2002.

[15] International Organization for Standardization. *ISO/IEC 19757-2: Document Schema Definition Languages (DSDL) - Part 2: Regular-grammar-based validation - RELAX NG.*

[16] World Wide Web Consortium (W3C). *XML Schema - W3C Recommendation.*

[17] International Organization for Standardization. *ISO/IEC 19757-3: Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron.*

[18] C. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, (19):17–37, 1982.

[19] D. Batory. The LEAPS algorithm. *Technical Report CS-TR-94-28*, 1994.

[20] Drools. `http://labs.jboss.com/portal/jbossrules`.

[21] EASYCOMP (IST Project 1999-14191) - Easy Composition in Future Generation Component Systems. `http://www.easycomp.org`.

[22] J. Raskin. *The humane interface: new directions for designing interactive systems.* Addison-Wesley Pearson Education, 2004.

[23] J. J. Garrett. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, 2005.

Aranea—Web Framework Construction and Integration Kit

Oleg Mürk
Dept. of Computer Science and Engineering,
Chalmers University of Technology,
SE-412 96 Göteborg, Sweden
oleg.myrk@gmail.com

Jevgeni Kabanov
Dept. of Computer Science,
University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
ekabanov@gmail.com

ABSTRACT

Currently there exist dozens of web controller frameworks that are incompatible, but at the same time have large portions of overlapping functionality that is implemented over and over again. Web programmers are facing limitations on code reuse, application and framework integration, extensibility, expressiveness of programming model and productivity.

In this paper we propose a minimalistic component model *Aranea* that is aimed at constructing and integrating server-side web controller frameworks in Java. It allows assembling most of available web programming models out of reusable components and patterns. We also show how to integrate different existing frameworks using Aranea as a common protocol. In its default configuration Aranea supports both developing sophisticated user interfaces using stateful components and nested processes as well as high-performance stateless components.

We propose to use this model as a platform for framework development, integration and research. This would allow combining different ideas and avoid reimplementing the same features repeatedly. An open source implementation of Aranea framework together with reusable controls, such as input forms and data lists, and a rendering engine are ready for real-life applications.

1. INTRODUCTION

During the last 10 years we have witnessed immense activity in the area of web framework design. Currently, there are more than 30 actively developed open source web frameworks in Java [10], let alone commercial products or other platforms like .NET and numerous dynamic languages. Not to mention in-house corporate frameworks that never saw public light. Many different and incompatible design philosophies are used, but even within one approach there are multiple frameworks that have small implementation differences and are consequently incompatible with each other.

The advantage of such a situation is that different approaches and ideas are tried out. Indeed, many very good ideas have been proposed during these years, many of which we will describe later in this paper. On a longer timescale the stronger (or better marketed) frameworks and approaches will survive, the weaker will diminish. However, in our opinion, such situation also has a lot of disadvantages.

1.1 Problem Description

First of all let's consider the problems of the web framework ecosystem from the viewpoint of application development. Framework user population is very fragmented as a result of having many incompatible frameworks with similar programming models. Each company or even project, is using a different web framework, which requires learning a different skill set. As a result, it is hard to find qualified work force for a given web framework. For the same reason it is even harder to reuse previously developed application code.

Moreover, it is sometimes useful to write different parts of the same application using different approaches, which might prove impossible, as the supporting frameworks are incompatible. Portal solutions that should facilitate integrating disparate applications provide very limited ways for components to communicate with each other. Finally, frameworks are often poorly designed, limiting expressiveness, productivity and quality.

System programmers face additional challenges. Creators of reusable components have to target one particular framework, consequently their market shrinks. Framework designers implement overlapping features over and over again, with each new feature added to each framework separately. Many useful ideas cannot be used together because they have been implemented in different frameworks.

We think that web framework market would win a lot if there were two or three popular platforms with orthogonal philosophies that would consolidate proponents of their approach. Application programmers would not have to learn a new web framework at the beginning of each project. Writing reusable components and application integration would be easier and more rewarding. Framework designers could try out new ideas much easier by writing extensions to the platform and targeting a large potential user-base.

1.2 Contributions

In this paper we will describe a component framework that we named *Aranea*. Aranea is written in Java and allows assembling server-side controller web frameworks out of reusable components and patterns. Aranea applications are pure Java and can be written without any static con-

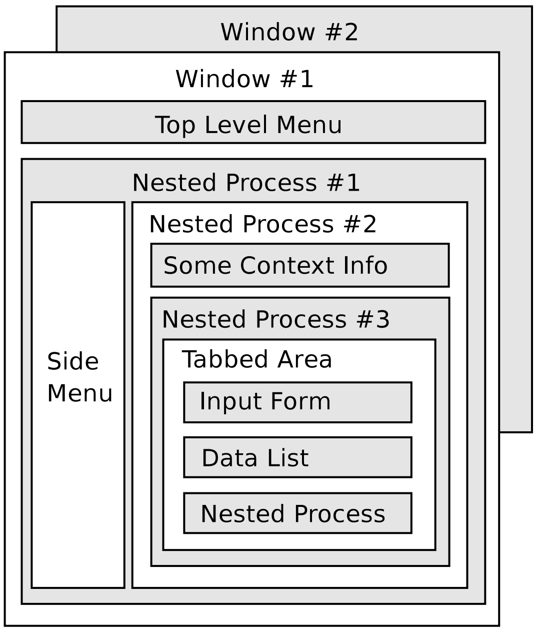


Figure 1: A sketch of a rich user interface.

figuration files. In Section 2 we describe our approach and motivation. We find that one of the strengths of this framework is its conceptual integrity—it has very few core concepts that are applied uniformly throughout the framework. The number of core interfaces is small, as is the number of methods in the interfaces. Components are easy to reuse, extend and test, because all external dependencies are injected into them. The details of the Aranea core abstractions are explained in Section 3.

In different configurations of Aranea components we can mimic principles and patterns of most existing server-side web controller frameworks as well as combine them arbitrarily. Possible configurations are described in Section 4. We concentrate on implementation of server-side controllers, but we also intend to support programming model where most of UI is implemented on the client-side and server-side contains only coarse-grained stateful components corresponding roughly to active use-cases.

Of particular interest is the configuration supporting programming model that allows expressing a rich user interface as a dynamic hierarchical composition of components that maintain call stacks of nested processes (we refer to such processes as *flows* further on). As an example of rich user interface at extreme, consider Figure 1: multiple interacting windows per user session, each window contains a stack of flows, flows can call nested flows that after completing return values, flows can display additional UI (side-menu, context information) even when a nested flow is executing, flows can contain tabbed areas, wizards, input forms, lists, other controls and even other flows.

Further, the framework facilitates both event-based and sequential programming (using continuations). The programming model is quite similar to the one used in the Smalltalk web framework Seaside [21], but has completely different implementation and is more general in terms of where sequential programming can be applied. This topic is discussed in Section 7.1 as one of extensions.

All web frameworks have to handle such aspects as configuration, security, error handling and concurrency. We explain how Aranea handles these issues in Section 5.

One of the most important differentiating factors of Aranea, is in our mind its ability to serve as a vehicle for integration of existing frameworks due to its orthogonal design. We discuss this topic in Section 6.

Finally, we see Aranea as a research platform. It it very easy to try out a new feature without having to write an entire web framework. A framework is assembled out of independent reusable components, so essentially everything can be reconfigured, extended or replaced. If Aranea becomes popular, writing a new component for Aranea would also mean a large potential user base.

Naturally, there are still numerous framework extensions to be made and further directions to be pursued. These are described in Section 7. Last, but not least, Aranea is based on great ideas originating from prior work of many people. When possible, we reference the original source of idea at the time of introducing it. In Section 8 we compare Aranea with some of the existing frameworks.

2. BACKGROUND

As we mentioned in the introduction, our aim is to support most of programming models and patterns available in the existing controller frameworks. We present here, unavoidably incomplete and subjective, list of profound ideas used in contemporary web controller frameworks.

The first important alternative is using stateless or reentrant components for high performance and low memory footprint, available in such frameworks as Struts [3] and WebWork [19].

Another important approach is using hierarchical composition of stateful non-reentrant components with event-based programming model, available in such frameworks as JSF [8], ASP.NET [4], Seaside [21], Wicket [20], Tapestry [5]. This model is often used for developing rich UI, but generally poses higher demands on server's CPU and memory.

The next abstraction useful especially for developing rich UI is nested processes, often referred to as *modal* processes, present for instance in such web frameworks as WASH [26], Cocoon [2], Spring Web Flow [18] and RIFE [14]. They are often referred to by the name of implementation mechanism—*continuations*. The original idea comes from Scheme [25], [22].

All of these continuation frameworks provide one top-level call-stack—essentially flows are like function calls spanning multiple web requests. A significant innovation can be found in framework Seaside [21], where continuations are combined with component model and call stack can be present at any level of component hierarchy.

Yet another important model is using asynchronous requests and partial page updates, coined Ajax [1]. This allows decreasing server-side state representation demands and increases responsiveness of UI. At the extreme, this allows creating essentially fat client applications with a sophisticated UI within browser.

We would also like to support different forms of metaprogramming such as domain specific language for describing

UI as a state machine in Spring Web Flow [18] and domain-driven design as implemented in Ruby on Rails [16] or RIFE/Crud [15]. This often requires framework support for dynamic composition and component run-time configuration.

3. CORE ABSTRACTIONS

Aranea framework is based on the abstraction of components arranged in a dynamic hierarchy and two component subtypes: services that model reentrant controllers and widgets that model non-reentrant stateful controllers. In this section we examine their interfaces and core implementation ideas. We omit some non-essential details for brevity.

3.1 Components

At the core of Aranea lies a notion of components arranged into a dynamic hierarchy that follows the *Composite* pattern extended with certain mechanisms for communication. This abstraction is captured by the following interface:

```
interface Component {
  void init(Environment env);
  void enable();
  void disable();
  void propagate(Message msg);
  void destroy();
}
```

A component is an entity that

- Has a life-cycle that begins with an `init()` call and ends with a `destroy()` call.
- Can be signaled to be disabled and then enabled again.
- Has an `Environment` that is passed to it by its parent or creator during initialization.
- Can propagate `Message`s to its children.

We imply here that a component will have a parent and may have children. Aranea actually implies that the component would realize a certain flavor of the *Composite* pattern that requires each child to have a unique identifier in relation to its parent. These identifiers can then be combined to create a full identifier that allows finding the component starting from the hierarchy root. Note that the hierarchy is not static and can be modified at any time by any parent.

The hierarchy we have arranged from our components so far is inert. To allow some communication between different components we need to examine in detail the notions of `Environment` and `Message`.

`Environment` is captured by the following interface:

```
interface Environment {GUI abstractions
  Object getEntry(Object key);
}
```

Environment is a discovery mechanism allowing children to discover services (named *contexts*) provided by their parents without actually knowing, which parent has provided it. Looking up a context is done by calling the environment `getEntry()` method passing some well-known context name as the key. By a convention this well-known name is the interface class realized by the context. The following example illustrates how environment can be used:

```
L10nContext locCtx = (L10nContext)
  getEnvironment().getEntry(L10nContext.class);
String message = locCtx.localize("message.key");
```

Environment may contain entries added by any of the current component ancestors, however the current component direct parent has complete control over the exact entries that the current component can discover. It can add new entries, override old ones as well as remove (or rather filter out) entries it does not want the child component to access. This is done by wrapping the grandparent `Environment` into a proxy that will allow only specific entries to be looked up from the grandparent.

`Message` is captured in the following interface:

```
interface Message {
  void send(Object key, Component comp);
}
```

While the environment allows communicating with the component parents, messages allow communicating with the component descendants (indirect children). `Message` is basically an adaptation of the *Visitor* pattern to our flavor of *Composite*. The idea is that a component `propagate(m)` method will just call message `m.send(...)` method for each of its children passing the message both their instances and identifiers. The message can then propagate itself further or call any other component methods.

It is easy to see that messages allow constructing both broadcasting (just sending the message to all of the components under the current component) and routed messages that receive a relative "path" from the current component and route the message to the intended one. The following example illustrates a component broadcasting some message to all its descendants (`BroadcastMessage` will call `execute` for all component under current):

```
Message myEvent = new BroadcastMessage() {
  public void execute(Component comp) {
    if (comp instanceof MyDataListener)
      ((MyDataListener) comp).setMyData(data);
  }
}
myEvent.send(null, rootComponent);
```

3.2 Services

Although component hierarchy is a very powerful concept and messaging is enough to do most of the communication, it is comfortable to define a specialized component type that is closer to the *Controller* pattern. We call this component `Service` and it is captured by the following interface:

```
interface Service extends Component {
  void action(
         Path path,
         InputData input,
         OutputData output
         );
}
```

`Service` is basically an abstraction of a reentrant controller in our hierarchy of components. The `InputData` and `OutputData` are simple generic abstractions over, correspondingly, a request and a response, which allow the controller to process request data and generate the response. The `Path` is an abstracted representation of the full path to the service from the root. It allows services to route the request to the one service it is intended for. Since service is also a component it can enrich the environment with additional contexts that can be used by its children.

3.3 Widgets

Although services are very flexible, they are not too comfortable for programming stateful non-reentrant components (GUI abstractions often being such). To do that we introduce the notion of a `Widget`, which is captured by the following interface:

```
interface Widget extends Service {
  void update(InputData data);
  void event(Path path, InputData input);
  void process();
  void render(OutputData output);
}
```

Widgets extend services, but unlike them widgets are usually stateful and are always assumed to be non-reentrant. The widget methods form a request-response cycle that should proceed in the following order:

1. `update()` is called on all the widgets in the hierarchy allowing them to read data intended for them from the request.

2. `event()` call is routed to a single widget in the hierarchy using the supplied `Path`. It allows widgets to react to specific user events.

3. `process()` is also called on all the widgets in the hierarchy allowing them to prepare for rendering whether or not the widget has received an event.

4. `render()` calls are not guided by any conventions. If called, widget should render itself (though it may delegate the rendering to e.g. template). The `render()` method should be idempotent, as it can be called arbitrary number of times after a `process()` call before an `update()` call.

Although widgets also inherit an `action()` method, it may not be called during the widget request-response cycle. The only time it is allowed is after a `process()` call, but before an `update()` call. It may be used to interact with a single widget, e.g. for the purposes of making an asynchronous request through Ajax [1].

Standard widget implementation allows setting event listeners that enable further discrimination between `action()/event()` calls to the same widget.

So far we called our components stateful or non-stateful without discussing the *persistence* of this state. A typical framework would introduce predefined scopes of persistence, however in Aranea we have very natural scopes for all our components—their lifetime. In Aranea one can just use the component object fields and assume that they will persist until the component is destroyed. If the session router is used, then the root component under it will live as long as the user session. This means that in Aranea state management is invisible to the programmer, as most components live as long as they are needed.

3.4 Flows

To support flows (nested processes) we construct a flow container widget that essentially hosts a stack of widgets (where only the top widget is active at any time) and enriches their environment with the following context:

```
interface FlowContext {
  void start(Widget flow, Handler handler);
  void replace(Widget flow);

  void finish(Object result);
  void cancel();
}
```

This context is available in standard widget implementation by calling `getFlowCtx()`. Its methods are used as follows:

- Flow `A` running in a flow container can start a nested flow `B` by calling `start(new B(...), null)`. The data passed to the flow `B` constructor can be thought as incoming parameters to the nested process. The flow `A` then becomes inactive and flow `B` gets initialized.

- When flow `B` is finished interacting with the user, it calls `finish(...)` passing the return value to the method. Alternatively flow `B` can call the `cancel()` method if the flow was terminated by user without completing its task and thus without a return value. In both cases flow `B` is destroyed and flow `A` is reactivated.

- Instead of finishing or canceling, flow `B` can also replace itself by a flow `C` calling `replace(new C(...))`. In such a case flow `B` gets destroyed, flow `C` gets initialized and activated, while flow `A` continues to be inactive. When flow `C` will finish flow `A` will get reactivated.

`Handler` callback interface is used when the calling flow needs to somehow react to the called flow finishing or canceling:

```
interface Handler {
  void onFinish(Object returnValue);
  void onCancel();
}
```

It is possible to use continuations to realize synchronous (blocking) semantics of flow invocation, as shown in the section 7, in which case the `Handler` interface is redundant.

4. FRAMEWORK ASSEMBLY

Now that we are familiar with the core abstractions we can examine how the actual web framework is assembled. First of all it is comfortable to enumerate the component types that repeatedly occur in the framework:

Filter A component that contains one child and chooses depending on the request parameters whether to route calls to it or not.

Router A component that contains many children, but routes calls to only one of them depending on the request parameters.

Broadcaster A component that has many children and routes calls to all of them.

Adapter A component that translates calls from one protocol to another (e.g. from service to a widget or from Servlet [6] to a service).

Container A component that allows some type of children to function by enabling some particular protocol or functionality.

Of course of all of these component types also enrich the environment and send messages when needed.

Aranea framework is nothing, but a hierarchy (often looking like a chain) of components fulfilling independent tasks. There is no predefined way of assembling it. Instead we show how to assemble frameworks that can host a flat namespace of reentrant controllers (á la Struts [3] actions), a flat namespace of non-reentrant stateful controllers (á la JSF [8] components) and nested stateful flows (á la Spring Web Flow [18]). Finally we also consider how to merge all these approaches in one assembly.

4.1 Reentrant Controllers

The first model is easy to implement by arranging the framework in a chain by containment (similar to pattern *Chain-of-Responsibility*), which starting from the root looks as follows:

1. Servlet [6] adapter component that translates the servlet `doPost()` and `doGet()` to Aranea service `action()` calls.

2. HTTP filter service that sets the correct headers (including caching) and character encoding. Generally this step consists of a chain of multiple filters.

3. URL path router service that routes the request to one of the child services using the URL path after servlet. One path will be marked as default.

4. A number of custom application services, each registered under a specific URL to the URL path router service that correspond to the reentrant controllers. We call these services *actions*.

The idea is that the first component object actually contains the second as a field, the second actually contains the third and so on. Routers keep their children in a `Map`. When `action()` calls arrive each component propagates them down the chain.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the default service.

- When custom services are invoked they can render an HTML response (optionally delegating it to a template) and insert into it URL paths of other custom services, allowing to route next request to them.

- A custom service may also issue an HTTP redirect directly sending the user to another custom service. This is useful when the former service performs some action that should not be repeated (e.g. money transfer).

Note that in this assembly `Path` is not used at all and actions are routed by the request URL.

Both filter and router services are stateful and reentrant. Router services could either create a new stateless action for each request (like WebWork [19] does) or route request to existing reentrant actions (like Struts [3] does). Router services could allow adding and removing (or enabling and disabling) child actions at runtime, although care must be taken to avoid destroying action that can be active on another thread.

We have shown above how analogues of Struts and WebWork actions fit into this architecture. WebWork interceptors could be implemented as a chain of filter services that decide based on `InputData` and `OutputData` whether to enrich them and then delegate work to the child service. There could be filter services both before action router and after. The former would be shared between all actions while the latter would be private for each action instance.

4.2 Stateful Non-Reentrant Controllers

To emulate the stateful non-reentrant controllers we will need to host widgets in the user session. To do that we assemble the framework as follows:

1. Servlet [6] adapter component.

2. Session router that creates a new service for each new session and passes the `action()` call to the associated service.

3. Synchronizing filter service that let's only one request proceed at a time.

4. HTTP filter service.

5. Widget adapter service that translates a service `action()` call into a widget `update()/event()/process()/render()` request-response cycle.

6. Widget container widget that will read from request the path to the widget that the event should be routed to and call `event()` with the correct path.

7. Page container widget that will allow the current child widget to replace itself with a new one.

8. Application root widget which in many cases is the login widget.

This setup is illustrated on Figure 2.

A real custom application would most probably have login widget as the application root. After authenticating login widget would replace itself with the actual root widget, which in most cases would be the application menu (which would also contain another page container widget as its child).

The menu would contain a mapping of menu items to widget classes (or more generally factories) and would start the appropriate widget in the child page container when the user clicks a menu item. The custom application widgets would be able to navigate among each other using the page context added by the page container to their environment.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the application root widget. If this is a new user session, a new session service will be created by the session router.

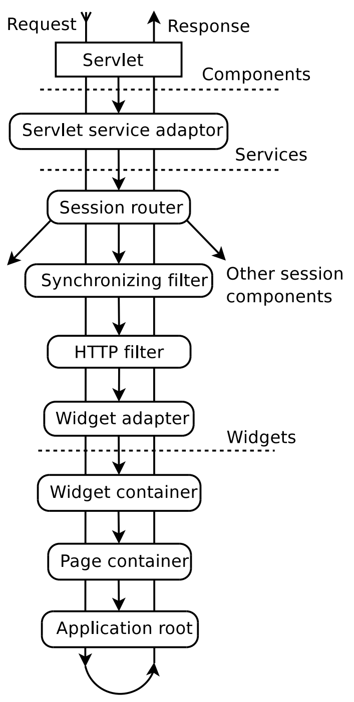


Figure 2: Framework assembly for hosting pages

- Only one request will be processed at once (due to synchronizing filter). This means that widget developers should never worry about concurrency.
- The widget may render a response, however it has no way of directly referencing other widgets by URLs. Therefore it must send all events from HTML to itself.
- Upon receiving an event the widget might replace itself with another widget (optionally passing it data as a constructor parameter) using the context provided by the page container widget. Generally all modifications of the widget hierarchy (e.g. adding/removing children) can only be done during event part of the request-response cycle.
- The hierarchy of widgets under the application root widget (e.g. GUI elements like forms or tabs) may be arranged using usual *Composite* widget implementations as no special routing is needed anymore.

In the real setup page container widget may be emulated using flow container widget that allows replacing the current flow with a new one.

Such an execution model is very similar to that of Wicket [20], JSF [8] or Tapestry [5] although these frameworks separate the pages from the rest of components (by declaring a special subclass) and add special support for markup components that compose the actual presentation of the page.

4.3 Stateful Non-Reentrant Controllers with Flows

To add nested processes we basically need only to replace the page container with a flow container in the previous model:

1. Servlet [6] adapter component.
2. Session router service.
3. Synchronizing filter service.
4. HTTP filter service.
5. Widget adapter service.
6. Widget container widget.
7. Flow container widget that will allow to run nested processes.
8. Application root flow widget which in many cases is the login flow.

The execution model here is very similar to the one outlined in Subsection 4.2. The only difference is that the application root flow may start a new subflow instead of replacing itself with another widget.

This model is similar to that of Spring WebFlow [18], although Spring WebFlow uses Push-Down Finite State Automaton to simulate the same navigation pattern and consequently it has only one top-level call stack. In our model call stacks can appear at any level of widget composition hierarchy, which makes our model considerably more flexible.

4.4 Combining the Models

It is also relatively easy to combine these models, modifying the model shown on figure 2 by putting a URL path router service before the session router, map the session router to a particular URL path and put a flow container in the end.

The combined model is useful, since reentrant stateless services allow to download files from database and send other semi-static data comfortably to the user. They can also be used to serve parts of the application that has the highest demand and thus load.

5. FRAMEWORK ASPECTS

Next we examine some typical web framework aspects and how they are realized in Aranea.

5.1 Configuration

The first aspect that we want to examine is *configuration.* We have repeated throughout the paper that the components should form a dynamic hierarchy, however it is comfortable to use a static configuration to wire the parts of the hierarchy that form the framework core.

To do that one can use just plain Java combining a hierarchy of objects using setter methods and constructors. But in reality it is more comfortable to use some configuration mechanism, like an IoC container. We use in our configuration examples Spring [17] IoC container and wire the components together as beans. Note that even such static configuration contains elements of dynamicity, since some components (á la root user session service) are wired not as instances, but via a factory that returns a new service for each session.

5.2 Security

The most common aspect of security that frameworks have to deal with is *authorization*. A common task is to determine, whether or not the current user has enough privileges to see a given page, component or GUI element. In many frameworks the pages or components are mapped to a particular URL, which can also be accessed directly by sending an HTTP request. In such cases it is also important to restrict the URLs accessible by the user to only those he is authorized to see.

When programming in Aranea using stateless re-entrant services they might also be mapped to particular URLs that need to be protected. But when programming in Aranea using widgets and flows (a stateful programming model) there is no *general* way to start flows by sending HTTP requests. Thus the only things that need protection are usually the menu (which can be assigned privileges per every menu item) and the active flow and widgets (which can only receive the events they subscribe to).

This simplifies the authorization model to checking whether you have enough privileges to start the flow *before* starting it. Since most use-cases should have enough privileges to start all their subflows it is usually enough to assign coarse-grained privileges to use-cases that can be started from the menu as well as fine-grained privileges for some particular actions (like editing instead of viewing).

5.3 Error Handling

When an exception occurs the framework must give the user (or the programmer) an informative message and also provide some recovery possibilities. Aranea peculiarity is that since an exception can occur at any level of hierarchy the informing and recovery may be specific to this place in the hierarchy. Default behavior for Aranea components is just to propagate the error up the hierarchy to the first exception handler component

For example it might be required to be able to cancel a flow that has thrown an exception and return back to the flow that invoked the faulty flow. A logical solution is to let the flow container (and other similar components) to handle their children's exceptions by rendering an informative error subpage instead in place of the flow. The error page can then allow canceling flows by sending events to the flow container.

With such approach when we have several flow containers on one HTML page, then if two or more flows under different containers fail, they will independently show error subpages allowing to cancel the particular faulty flows. Note also that such approach will leave the usual navigation elements like menus intact, which will allow the user to navigate the application as usual.

5.4 Concurrency

Execution model of Aranea is such that each web request is processed on one Java thread, which makes system considerably easier to debug. By default Aranea does not synchronize component calls. It does, however, protect from trying to destroy a working component. If a service or widget currently in the middle of some method call will be destroyed, the destroyer will wait until it returns from the call. To protect from deadlock and livelock, after some time the lock will be released with a warning.

When we want to synchronize the actual calls (as we need for example with widgets) we can use the synchronizing service that allows only one `action()` call to take place simultaneously. This service can be used when configuring the Aranea framework to synchronize calls on e.g. browser window threads. This allows to program assuming that only one request per browser window is processed at any moment of time. Note that widgets should *always* be behind a synchronizing filter and cannot process concurrent calls.

6. INTEGRATION SCENARIOS

In this section we describe our vision of how web controller frameworks could be integrated with Aranea or among each other. In practice, we have so far integrated Aranea only with one internal framework with stateful Portlet-like [12] components, where Aranea components were hosted within the latter framework, but we are considering integrating with such frameworks as Wicket [20], JSF [8], Tapestry [5], Spring WebFlow [18], Struts [3], and WebWork [19].

Integration is notorious for being hard to create generalizations about. Each integration scenario has its own set of specialized problems and we find that this article is not the right place to write about them. For this reason we keep this section intentionally very abstract and high-level and try to describe general principles of web controller framework integration without drowning in implementation details.

In the following we assume that depending on their nature it is possible to model components of frameworks we want to integrate as one of:

- *service-like*—reentrant and/or stateless component[1],
- *widget-like*—non-reentrant stateful component.

Note that both notions consist of two contracts: interface of component and contract of the container of the component.

In our abstraction we have essentially the following integration scenarios:

- service-service,
- service-widget,
- widget-service,
- widget-widget.

Here, for instance, "service-widget" should be read as: "*service*-like component of framework X containing *widget*-like component of framework Y". In homogeneous (i.e. service-service and widget-widget) integration scenarios one has to find a mapping between service (resp. widget) interface methods invocations of two frameworks. Although we do not find this mapping trivial, there is little we can say without considering specialized details of particular frameworks. However, our experience shows that, thanks to minimalistic and orthogonal interfaces and extensibility of Aranea, the task becomes more tractable than with other monolithic frameworks. We now concentrate on heterogeneous cases of server-widget and widget-service integration. They can also occur within Aranea framework itself, but are more typical when disparate frameworks using different programming models are integrated.

In service-widget scenario, generally, each web request is processed by some service and then the response is rendered

[1]Note that Servlets [6] are, for instance, service-like components.

by possibly different service, whereas both services can be reentrant and/or stateless. As a result, such services cannot host themselves the widgets whose life-time spans multiple requests handled by different services. Consequently, widget instances should be maintained in stateful widget container service(s) with longer life-span. At each request such services would call `update()` and `event()` methods of the contained widgets. Widgets would be instantiated by services processing the request and rendered using `render()` method by services generating the response. Each service processing a request should explicitly decide which widgets are to be kept further, all the rest are to be destroyed (within current session). As the services are generally reentrant, it is important to exclude concurrent access to the widgets belonging to the same session. The simplest solution is to synchronize on session at each web request that accesses widgets.

In widget-service scenario, services should be contained in service container widgets in the position within widget hierarchy most suitable for rendering the service. On widget `update()`, the data entitled for the contained service should be memorized. On widget `render()` the memorized data should be passed to the `action()` method of contained service to render the response. If the service responds with redirection, which means that the request should not be rerun, the service should be replaced with the service to which the redirection points. After that and on all subsequent renderings the `action()` method of the new service should be called with redirection parameters.

Coming back to not-so-abstract reality, when integrating frameworks the following issues should be handled with care:

- How data is represented in the web request and how output of multiple components coexists in the generated web response.
- Namespaces (e.g. field identifiers in web request) of contained components should not mix with the namespace of container components, which in general means appending to the names a prefix representing location of contained component within the container.
- State management, especially session state history management (browser's back, forward, refresh and new window navigation commands) and keeping part of the state on the client, should match between integrated components. We explore this topic further in Subsection 7.2.
- A related issue to consider is view integration. Many web frameworks support web components that are tightly integrated with some variant of templating. Consequently it is important that these templating technologies could be intermixed easily.

Incompatibilities in these aspects lead to a lot of mundane protocol conversion code, or even force modifying integrated components and/or frameworks.

Generalized solutions to these issues could be standardized as *Aranea Protocol*. As compared to such protocol, current Aranea Java interfaces are relatively loose—i.e. functionality can be considerably customized by using `Message` protocol and extending core interfaces (`InputData`, `OutputData`, `Component`) with new subintefaces.

Altogether, we envision the following integration scenarios with respect to Aranea:

Guest Aranea components (resp. services or widgets) are hosted within components of framework X that comply to the Aranea component (resp. service or widget) container contract.

Host Aranea components host components (resp. service-like or widget-like) of framework X through an adapter component that wraps framework X components into Aranea component (resp. service or widget) interface.

Protocol Framework X components provide Aranea component (resp. service or widget) container contract that hosts framework Y components wrapped into Aranea component (resp. service or widget) interface using an adapter component.

7. EXTENSIONS AND FUTURE WORK

In this section we discuss important functionality that is not yet implemented in Aranea. In some cases we have very clear idea how to do it, in other cases our understanding is more vague.

7.1 Blocking Calls and Continuations

Consider the following very simple scenario:

1. When user clicks a button, we start a new subflow.
2. When this subflow eventually completes we want to assign its return value to some text field.

In event-driven programming model the following code would be typical:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    Handler handler = new Handler() {
      void onFinish(Object result) {
        field.setText((String)result);
      }
    }
    getFlowCtx().
      start(new SubFlow(), handler);
  }
}
button.addOnClickListener(listener);
```

What strikes here is the need to use multiple event listeners, and as a result writing multiple anonymous classes that are clumsy Java equivalent of syntactical closures. What we would like to write is:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    String result = (String)getFlowCtx().
             call(new SubFlow());
    label.setText(result);
  }
}
button.addOnClickListener(listener);
```

What happens here is that flow is now called using blocking semantics.

Typically blocking behavior is implemented by suspending executed thread and waiting on some concurrency primitive like semaphore or monitor. The disadvantage of such solution is that operating system threads are expensive, so using an extra thread for each user session would be a major overkill—most application servers use a limited pool of worker threads that would be exhausted very fast. Besides, threads cannot be serialized and migrated to other cluster nodes. A more conceptual problem is that suspended thread contains information regarding processing of the whole web request, whereas it can be woken up by a different web request. Also, in Java blocking threads would retain ownership of all monitors.

In [25] and [22] *continuations* were proposed to solve the blocking problem in web applications, described above. Continuation can be thought of as a lightweight snapshot of thread's call stack that can be resumed multiple times. There still remains the problem that both thread and continuation contain information regarding processing of the whole request, but can be woken up by a different web request.

To solve this problem *partial continuations* [23] can be used. Essentially, the difference is that the snapshot of call stack is taken not from the root, but starting from some stack frame that we will call *boundary.* In case of Aranea, the boundary will be the stack frame of event handler invocation that may contain blocking statements. So in case of our previous example the boundary will be invocation of method `onClick()`. When we need to wait for an event, the following should be executed:

1. Take current partial continuation,
2. Register it as an event handler,
3. Escape to the boundary.

Similar approach can be also applied to services though mimicking such frameworks as Cocoon [2] and RIFE [14]. We'd like to stress that by applying continuations to widget event handlers we can create a more powerful programming model because there can be simultaneous linear flows at different places of the same widget hierarchy, e.g. in each flow container. This programming model is similar to that of Smalltalk web framework Seaside [21] that uses continuations to provide analogous blocking call semantics of flows, but not event handlers in general.

Java does not have native support for continuations, but luckily there exists experimental library [7] that allows suspending current partial continuation and later resuming it. Aranea currently does not have implementation of this approach, however, it should be relatively easy to do that. Event handlers containing blocking statements should be instrumented with additional logic denoting continuation boundaries. We could use e.g. AspectJ [24] to do that.

Altogether we view blocking calls as a rather easily implementable syntactic sugar above the core framework. At the same time we find that combining event-based and sequential programming in a component framework is a very powerful idea because different parts of application logic can be expressed using the most suitable tool.

7.2 State Management

We have also solutions to the following problems related to state management:

- *Optimizing memory consumption*—in high performance applications low memory consumption of session state representation is essential. The interface of `Component` has methods `disable()` and `enable()` that allow releasing all unnecessary resources when disabled.
- *Client-side state*—part of session state can be kept within web response that later becomes web request or within a cookie [13]. This also allows reducing server-side memory consumption.
- *Navigation history*—supporting (or sensibly ignoring) browser's back, forward, refresh and new window navigation commands. This can be useful both for usability, decreasing server-side state representation or for integrating with other frameworks that rely on browser's navigation commands as the only navigation mechanism.

7.3 Integration and Portals

It is important to note that so far we have described only applications that are configured before deployment and work within one Java virtual machine (or homogeneous cluster). There are portal applications that would benefit from dynamic reconfiguration and using widgets or flows deployed to another environment. The latter could happen for multiple reasons such as using a different platform (like .NET), co-location of web application with database or just administrative reasons.

One possible approach is to integrate with Portlet [12] specification together with remote integration protocol WSRP [9]. Unfortunately portlets cannot be composed into hierarchies and have many limitations on how they can communicate with each other. There is also no notion of nested process in portlets. Finally, portal implementations that we are aware of allow reconfiguring portals only by redeployment.

It should be easy to assemble out of Aranea components a portal application that would contain multiple pre-packaged applications, communicating with each other, but the configuration would have to be read on deployment. One further direction is to integrate Aranea with some component framework allowing dynamic reconfiguration, such as OSGi [11].

Another related direction is to develop a remote integration protocol that would allow creating a widget that would be a proxy to a widget located in another environment. One important issue would be minimizing the number of roundtrips.

7.4 Fat Client

Lately, more and more web applications started using asynchronous requests to update parts of the page without resubmitting and refreshing the whole page. Some applications even implement most of UI logic on the client-side and use web server essentially as a container for the business layer. The enabling technology is called Ajax [1] and is essentially a small API that allows sending web requests to the server. We think that this trend will continue and in future most application will use this approach to a varying extent.

The first option is when UI logic is still implemented on the server-side, but in order to make web pages more responsive sometimes ad-hoc asynchronous requests are used to update page structure without refreshing the whole page. This can be accomplished in Aranea using either messages or the fact that widgets extend services and consequently have `action(input,output)` method. Within widget, some kind of simple event handling logic could be implemented.

Another option is when all UI implemented on the client-side within browser and server-side controller acts essentially as a business layer. Although business layer is often stateless, we find that Aranea could be used to create a coarse-grained server-side representation of UI state, essentially representing activated use-cases, modeled most naturally as flows. Client-side UI would we able to only execute commands making sense in the context of current server-side UI state. Such approach is very convenient for enforcing complex stateful authorization rules and data validation would have to be performed on the server-side in any case.

8. RELATED WORK

As it was mentioned before, Aranea draws its ideas from multiple frameworks such as Struts [3], WebWork [19], JavaServer Faces [8], ASP.NET [4], Wicket [20], Tapestry [5], WASH [26], Cocoon [2], Seaside [21], Spring Web Flow [18], and RIFE [14]. When possible we have referenced the original source of idea at the moment of introducing it.

Although we were not aware of Seaside [21] when developing this framework, we have to acknowledge that rich UI programming interface of widgets and flows is almost identical with programming interface of Seaside, but the design of Seaside differs a lot and it is not intended as a component model for web framework construction and integration.

9. ACKNOWLEDGEMENTS

Development of Aranea implementation has been supported by Webmedia, Ltd. We are grateful to Maksim Boiko, who prototyped the implementation in his bachelor thesis and Konstantin Tretyakov, who provided valuable input as well as help with developing the presentation module. This work was partially supported by Estonian Science Foundation grant No. 6713.

10. SUMMARY

In this paper we have motivated and described a component model for assembling web controller frameworks. We see it as a platform for framework development, integration and research.

There exists an open source implementation of Aranea framework available at `http://araneaframework.org/`. It is bundled together with reusable controls, such as input forms and data lists and advanced JSP-based rendering engine. This framework has been used in real projects and we find it ready for production use. Interested reader can also find at this address an extended version of this article with many details that had to be omitted here.

11. REFERENCES

[1] Ajax. Wikipedia encyclopedia article available at `http://en.wikipedia.org/wiki/AJAX`.

[2] Apache Cocoon project. Available at `http://cocoon.apache.org/`.

[3] Apache Struts project. Available at `http://struts.apache.org/`.

[4] ASP.NET. Available at `http://asp.net/`.

[5] Jakarta Tapestry. Available at `http://jakarta.apache.org/tapestry/`.

[6] Java Servlet 2.4 Specification (JSR-000154). Available at `http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html`.

[7] The Javaflow component, Jakarta Commons project. Available at `http://jakarta.apache.org/commons/sandbox/javaflow/index.html`.

[8] JavaServer Faces technology. Available at `http://java.sun.com/javaee/javaserverfaces/`.

[9] OASIS Web Services for Remote Portlets. Available at `www.oasis-open.org/committees/wsrp/`.

[10] Open source web frameworks in Java. Available at `http://java-source.net/open-source/web-frameworks`.

[11] OSGi Service Platform. Available at `http://www.osgi.org/`.

[12] Portlet Specification (JSR-000168). Available at `http://www.jcp.org/aboutJava/communityprocess/final/jsr168/`.

[13] RFC 2109 - HTTP State Management Mechanism. Available at `http://www.faqs.org/rfcs/rfc2109.html`.

[14] RIFE. Available at `http://rifers.org/`.

[15] RIFE/Crud. Available at `http://rifers.org/wiki/display/rifecrud/`.

[16] Ruby on Rails. Available at `http://www.rubyonrails.org/`.

[17] Spring. Available at `http://springframework.org`.

[18] Spring Web Flow. Available at `http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/`.

[19] WebWork, OpenSymphony project. Available at `http://struts.apache.org/`.

[20] Wicket. Available at `http://wicket.sourceforge.net/`.

[21] S. Ducasse, A. Lienhard and L. Renggli. Seaside — a multiple control flow web application framework. *ESUG 2004 Research Track*, pages 231–257, September 2004.

[22] P. T. Graunke, S. Krishnamurthi, V. der Hoeven and M. Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming (ESOP 2001)*, 2001.

[23] R. Hieb, K. Dybvig and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. Project web site: `http://www.eclipse.org/aspectj/`.

[25] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33, 2000.

[26] P. Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. Available at `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/`.

Session G
Short Papers

Typeless programming in Java 5.0

Martin Plümicke
University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15, D–72160 Horb
m.pluemicke@ba-horb.de

Jörg Bäuerle
AWM Media
International IT Department
P.O. Box 4006, Worthing BN13 1AP, UK
Joerg.Baeuerle@gmx.net

ABSTRACT

With the introduction of Java 5.0 [9] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.

Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not. Furthermore there are methods whose principle types would be intersection types. But intersection types are not implemented in Java 5.0. This means that Java 5.0 methods often don't have the principle type which is contradictive to the OOP-Principle of writing re-usable code.

This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principle types.

We implement the algorithm in Java using the observer design pattern.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms

Algorithms, Theory

Keywords

Code generation, language design, program design and implementation, type inference, type system

```
class Matrix extends Vector<Vector<Integer>> {
  Matrix mul(Matrix m){
    Matrix ret = new Matrix();
    int i = 0;
    while(i <size()) {
      Vector<Integer> v1 = this.elementAt(i);
      Vector<Integer> v2 = new Vector<Integer>();
      int j = 0;
      while(j < v1.size()) {
        int erg = 0;
        int k = 0;
        while(k < v1.size()) {
          erg = erg + v1.elementAt(k)
                  * m.elementAt(k).elementAt(j);
          k++; }
        v2.addElement(new Integer(erg));
        j++; }
      ret.addElement(v2);
      i++; }
    return ret; }
}
```

Figure 1: The class `Matrix`

1. INTRODUCTION

In this paper we present a type inference algorithm for a core language of Java 5.0. Type inference means that we can implement Java 5.0 programs without type annotations of method parameters and return types and of local variables. The type inference system calculates them automatically.

Let us consider an example. The class `Matrix` (fig. 1) extends `Vector<Vector<Integer>>`. `Matrix` has the method `mul`, which implements the multiplication of matrices. The parameters, the return type, and the local variables are explictly typed (underlined in fig. 1). If we consider the type annotations more accurately, we will observe that there is more than one possibility to give correct type annotations. The return type, the type annotation of `m`, and the type annotation of `ret` are not unambiguous. The type `Vector<Vector<Integer>>` would also be a correct type annotation. This means, considering the type of the method `mul`, that it has beside the type `Matrix` $\rightarrow$ `Matrix`, the type `Vector<Vector<Integer>>` $\rightarrow$ `Vector<Vector<Integer>>`, and all mixtures. The fact that a method has more than one type is called intersection type. In Java 5.0 no annotations of intersection types are allowed.

The idea of Java 5.0 type inference is to omit these type annotations. The system automatically calculates the principle intersection type of the methods.

The type inference discipline arose in functional programming languages (e.g. Haskell [10] or SML [14]). A basic paper for type inference including an algorithm and the definition of principle types is given by [3]. A method's principle type is a type, where all correct types of the method are derivable from. In section 4.3 we give a detailed definition.

Many papers on type inference have been published. Some papers consider type inference in object–oriented languages. In [15, 11] type inference systems for a language oriented at Smalltalk [8] are given. In this approach the idea is to collect type constraints, build a uniform set, and solve them by a fixed point derivation. In [5] similar as in [1] types are defined as a pair of a conventional type and a set of type constraints. Finally, the constraints are satisfied, if possible. These type systems differ from the Java 5.0 type system.

In [4] and [6] a refactoring is presented that replaces raw references to generic library classes with parametrized references. For this the type parameters of the raw type annotations must be inferred. The main difference to our approach is, that in our approach no type annotation, must be given and the algorithm infers the whole type.

The type system of polymorphically order-sorted types, which is considered for a logical language [18] and for a functional object–oriented language OBJ–P [16], is very similar to the Java 5.0 type system. Therefore our approach is oriented at polymorphically order-sorted types.

The paper is organized as follows. In the second section we formally describe the Java 5.0 type system. In the third section, we give a unification algorithm for a subset of the Java 5.0 types. The unification algorithm is the base of the type inference algorithm. In the fourth section we present the type inference algorithm itself, illustrate the algorithm by an example, and consider the properties of the algorithm. Finally in the sixth section we present the implementation. We have done the implementation in Java using the observer design pattern. We close the paper with a conclusion and an outlook.

2. JAVA 5.0 TYPE SYSTEM

As a base for the type inference algorithm we have to give a formal definition of the Java 5.0 type system. First we introduce the simple types. Fields, method parameters, return types of methods, and local variables are annotated by simple types. Then, from the *extends* relation we derive the subtyping relation. Furthermore, we define a relation *finite closure*, which is necessary for the unification algorithm (section 3). Finally, we introduce function types, which represent the types of methods.

The simple types are built as a set of terms over a finite rank alphabet of the class/interface names, a set of unbounded respectively bounded type variables, and unbounded respectively bounded wildcards.

Definition 1. Let JC be a set of declared Java 5.0 classes and $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$ the finite rank alphabet of class/interface names indexed by its respective number of parameters. Let TV be a set of unbounded type variables. Furthermore, let $T_\Theta(TV)$ be the set of (type) terms over Θ and TV.

Then, the set of *simple types* $\mathsf{SType}_\Theta(BTV)$ is the smallest set satisfying the following conditions:

- $T_\Theta(TV) \subseteq \mathsf{SType}_\Theta(BTV)$
- For each intersection of simple types $ty = ty_1 \& \ldots \& ty_n$
$$BTV^{(ty)} \subseteq \mathsf{SType}_\Theta(BTV)$$
where $BTV^{(ty)}$ contains all type variables, bounded by ty. A bounded type variable is denoted by $a|_{ty}$.
- For $\theta_i \in \mathsf{SType}_\Theta(BTV) \cup \{\,?\,\} \cup \{\,?\,\mathtt{extends}\ \tau \mid \tau \in \mathsf{SType}_\Theta(BTV)\} \cup \{\,?\,\mathtt{super}\ \tau \mid \tau \in \mathsf{SType}_\Theta(BTV)\}$
and $C \in \Theta^{(n)} : C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \in \mathsf{SType}_\Theta(BTV)$.

The inheritance hierarchy consists of two different relations: The "*extends* relation" (in sign $<$) is explicitly defined in Java 5.0 programs by the *extends* respectively the *implements* declarations. The "subtyping relation" is then built as the reflexive, transitive, and instantiating closure of the *extends* relation.

Definition 2. Let JC be a set of declared Java 5.0 classes, $\mathsf{SType}_\Theta(BTV)$ the corresponding set of simple types, and $\leq$ the corresponding extends relation. The *subtyping relation* $\leq^*$ is given as the smallest ordering satisfying the following conditions:

- if $(\theta, \theta') \in \mathsf{SType}_\Theta(BTV) \times \mathsf{SType}_\Theta(BTV)$ is an element of the reflexive and transitive closure of $\leq$ then $\theta \leq^* \theta'$.
- if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ for all substitutions σ_1, σ_2, which satisfy for each type variable a of θ_2 one of the following conditions:
  - $\sigma_1(a) = \sigma_2(a)$
  - $\sigma_1(a) = \theta$ and $\sigma_2(a) = \,?$
  - $\sigma_1(a) = \theta$ and $\sigma_2(a) = \,?\,\mathtt{extends}\,\theta'$ and $\theta \leq^* \theta'$
  - $\sigma_1(a) = \,?\,\mathtt{super}\ \theta$ and $\sigma_2(a) = \theta'$ and $\theta \leq^* \theta'$
- $a \leq^* \theta'$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ where $\exists \theta_i : \theta_i \leq^* \theta'$.

It is surprising that the condition for σ_1 and σ_2 in the first subitem is not $\sigma_1(a) \leq^* \sigma_2(a)$, but $\sigma_1(a) = \sigma_2(a)$. This is necessary to get a sound type system (cp. [9]).

Furthermore, we declare an ordering on the set of type terms which we call the finite closure of the *extends* relation. This ordering is necessary for the type unification algorithm (section 3).

Definition 3. The *finite closure* $\mathbf{FC}(\leq)$ is the reflexive and transitive closure of pairs in the *extends* relation $C(a_1 \ldots a_n) \leq D(\theta_1, \ldots, \theta_m)$, where the a_i are type variables and the θ_i are real type terms.

Now we give an example to illustrate the abstract definitions.

Example 1. Let the following Java 5.0 program be given.

```
abstract class AbstractList<a> implements List<a>{...}

class Vector<a> extends AbstractList<a> {...}

class Matrix<a> extends Vector<Vector<a>> {...}
```

(reduce1) $$\frac{Eq \cup \{\, C{<}\theta_1, \ldots, \theta_n{>} \lessdot D{<}\theta'_1, \ldots, \theta'_n{>} \,\}}{Eq \cup \{\, \theta_{\pi(1)} \doteq \theta'_1, \ldots, \theta_{\pi(n)} \doteq \theta'_n \,\}}$$
where
- $C{<}a_1, \ldots, a_n{>} \leq^* D{<}a_{\pi(1)}, \ldots, a_{\pi(n)}{>}$
- $\{\, a_1, \ldots, a_n \,\} \subseteq TV$
- π is a permutation

(reduce2) $$\frac{Eq \cup \{\, C{<}\theta_1, \ldots, \theta_n{>} \doteq C{<}\theta'_1, \ldots, \theta'_n{>} \,\}}{Eq \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

(erase) $$\frac{Eq \cup \{\, \theta \doteq \theta' \,\}}{Eq} \quad \theta = \theta'$$

(swap) $$\frac{Eq \cup \{\, \theta \doteq a \,\}}{Eq \cup \{\, a \doteq \theta \,\}} \quad \theta \notin TV,\ a \in TV$$

(adapt) $$\frac{Eq \cup \{\, D{<}\theta_1, \ldots, \theta_n{>} \lessdot D'{<}\theta'_1, \ldots, \theta'_m{>} \,\}}{Eq \cup \{\, D'{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m{>}[a_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \lessdot D'{<}\theta'_1, \ldots, \theta'_m{>} \,\}}$$
where there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with
- $(D{<}a_1, \ldots, a_n{>} \leq^* D'{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m{>}) \in \mathbf{FC}(\leq)$

(subst) $$\frac{Eq \cup \{\, a \doteq \theta \,\}}{Eq[a \mapsto \theta] \cup \{\, a \doteq \theta \,\}}$$
where
- a occurs in Eq but not in θ

Figure 2: Java 5.0 type unification

The following type term pairs are elements of the subtyping relation $\leq^*$:

$\texttt{Vector<Vector<a>>} \leq^* \texttt{Vector<Vector<a>>}$,
$\texttt{Vector<Vector<a>>} \leq^* \texttt{AbstractList<Vector<a>>}$,
$\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}$,
$\texttt{Matrix<a>} \leq^* \texttt{AbstractList<Vector<a>>}$.

But $\texttt{Vector<Vector<a>>} \not\leq^* \texttt{Vector<AbstractList<a>>}$ which follows from the soundness of the Java 5.0 type system.

The finite closure $\mathbf{FC}(\leq)$ is given as the reflexive and transitive closure of $\{\, \texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>} \leq^* \texttt{AbstractList<Vector<a>>} \leq^* \texttt{List<Vector<a>>} \,\}$.

We complete the Java 5.0 type system with the following definition.

Definition 4. Let $\mathsf{SType}_\Theta(BTV)$ be a set of simple types of given Java 5.0 classes. The respective set of *Java 5.0 types* $\mathsf{Type}(\mathsf{SType}_\Theta(BTV))$ is the smallest set with the following properties:

1. For the considered **base types** holds
   $\{\, \texttt{Integer}, \texttt{Boolean}, \texttt{Char} \,\} \subseteq \mathsf{Type}(\mathsf{SType}_\Theta(BTV))$.
2. $\mathsf{SType}_\Theta(BTV) \subseteq \mathsf{Type}(\mathsf{SType}_\Theta(BTV))$ (**simple type**).
3. If for $0 \leqslant i \leqslant n$: $\theta_i \in (\mathsf{SType}_\Theta(BTV) \cup \mathsf{basetype})$ then $\theta_1 \times \ldots \times \theta_n \to \theta_0 \in \mathsf{Type}(\mathsf{SType}_\Theta(BTV))$ (**function type**).
4. If $ty_1, ty_2 \in \mathsf{Type}(\mathsf{SType}_\Theta(BTV))$, then $ty_1 \& ty_2 \in \mathsf{Type}(\mathsf{SType}_\Theta(BTV))$ (**intersection type**).

The *base types* and the *simple types* describe types of fields, types of methods' parameters, return types of methods, and types of local variables. Finally the types of the methods are given as intersections of *function types*. The intersections are necessary to describe the principle type of a method. If a method has an intersection type, this means that more than one type is inferable for the code.

We do not consider *raw types* as they are only necessary to use older Java programs (Version $\leq$ 1.4). The behavior of raw types can be simulated by using the corresponding parameterized types, where all arguments are instantiated by `Object`.

3. TYPE UNIFICATION

The basis of the type inference algorithm is the type unification. The *type unification problem* is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

σ is called a unifier of θ_1 and θ_2. In the following we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified.

Our type unification algorithm is based on the unification algorithm of Martelli and Montanari [13]. The main difference is, that in the original unification a unifier is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$. This means that a pair $a \doteq \theta$ determines that the unifier substitutes a by the term θ. In contrast a pair $a \lessdot \theta$ respectively $\theta \lessdot a$ leads to multiple correct substitutions. All type terms smaller than θ and greater than θ respectively are correct substitutions for a. This means that there are multiple unifiers.

Now, we give the type unification algorithm. We restrict the type terms to terms without bounded type variables and without wildcards. We denoted this subset of $\mathsf{SType}_\Theta(BTV)$ in definition 1 by $T_\Theta(TV)$.

The algorithm itself is given in seven steps:

1. For each pair $a \lessdot \theta$ a set of pairs is built, which contains for all substypes $\overline{\theta}$ of θ the pair $a \doteq \overline{\theta}$.
2. For each pair $\theta \lessdot a$ a set of pairs is built, which contains for all supertypes θ' of θ the pair $a \doteq \theta'$.
3. The cartesian product of the sets from step 1 and 2 is built.
4. Repeated application of the rules *reduce1*, *reduce2*, *erase*, *swap*, and *adapt* (fig. 2) to each set of type term pairs.
5. Application of the rule *subst* (fig. 2) to each set of type term pairs.
6. For all changed sets of type terms start again with step 1.
7. Summerize all results

For more details about the Java 5.0 type unification see in [17].

Example 2. Let the subtyping relation and the finite closure be given as in example 1.

$$
\begin{array}{lcl}
\mathit{Source} & := & (\mathit{class} \mid \mathit{interface})* \\
\mathit{class} & := & \textsf{Class}(\mathit{stype}, [\ \textsf{extends}(\ \mathit{stype}\ ),]\ [\ \textsf{implements}(\ \mathit{stype}+\ ),]\mathit{IVarDecl}*, \mathit{Method}*) \\
\mathit{interface} & := & \textsf{interface}(\mathit{stype}, [\textsf{extends}(\ \mathit{stype}\ ),]\mathit{MHeader}*) \\
\mathit{IVarDecl} & := & \textsf{InstVarDecl}(\ \mathit{stype}, \mathit{var}\ ) \\
\mathit{MHeader} & := & \textsf{MethodHeader}(\ \mathit{mname}, \mathit{stype}, (\mathit{var}, \mathit{stype})*\ ) \\
\mathit{Method} & := & \textsf{Method}(\ \mathit{mname}, \underline{[\mathit{stype}]}, (\mathit{var}\underline{[, \mathit{stype}]})*, \mathit{block}\ ) \\
\mathit{block} & := & \textsf{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block} \mid \textsf{Return}(\ \mathit{expr}\ ) \mid \textsf{while}(\ \mathit{expr}, \mathit{block}\ ) \mid \textsf{LocalVarDecl}(\ \mathit{var}\underline{[, \mathit{stype}]}\ ) \mid \textsf{If}(\ \mathit{expr}, \mathit{block}[, \mathit{block}]\ ) \\
 & \mid & \mathit{stmtexpr} \\
\mathit{stmtexpr} & := & \textsf{Assign}(\ \mathit{var}, \mathit{expr}\ ) \mid \textsf{New}(\ \mathit{stype}, \mathit{expr}*\ ) \mid \textsf{NewArray}(\ \mathit{stype}, \mathit{expr}\ ) \mid \textsf{MethodCall}(\ [\mathit{expr},]f, \mathit{expr}*\ ) \\
\mathit{expr} & := & \mathit{stmtexpr} \mid \textsf{this} \mid \textsf{super} \mid \textsf{LocalOrFieldVar}(\ \mathit{var}\ ) \mid \textsf{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \mid \textsf{ArrayAcc}(\ \mathit{expr}, \mathit{expr}\ ) \\
 & \mid & \textsf{Add}(\ \mathit{expr}, \mathit{expr}\ ) \mid \textsf{Minus}(\ \mathit{expr}, \mathit{expr}\ ) \mid \textsf{Mul}(\ \mathit{expr}, \mathit{expr}\ ) \mid \textsf{Div}(\ \mathit{expr}, \mathit{expr}\ ) \mid \textsf{Mod}(\ \mathit{expr}, \mathit{expr}\ ) \\
 & \mid & \textsf{Not}(\ \mathit{expr}\ ) \mid \textsf{And}(\ \mathit{expr}, \mathit{expr}\ ) \mid \textsf{Or}(\ \mathit{expr}, \mathit{expr}\ )
\end{array}
$$

Figure 3: The Java 5.0 core language

We apply the unification algorithm to

$$\{\ \texttt{Matrix<b>} \lessdot \texttt{Vector<Vector<List<Object>>>},\ \texttt{a} \lessdot \texttt{b}\ \}$$

In the first three steps nothing happens.
In step 4 we get by the *adapt* rule

$$\{\ \{\ \texttt{Vector<Vector<b>>} \doteq \texttt{Vector<Vector<List<Object>>>},\ \texttt{a} \lessdot \texttt{b}\ \}\ \},$$

as ($\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}) \in \mathbf{FC}(\ \leq\ )$. Then the *reduce2* rule leads to: $\{\ \{\ \texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \lessdot \texttt{b}\ \}\ \}$. In step 5 the *subst* rule leads to

$$\{\ \{\ \texttt{b} \doteq \texttt{List<Object>},\ \texttt{a} \lessdot \texttt{List<Object>}\ \}\ \}$$

With the again application of the first three steps we get finally:

$$\{\ \{\ \texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{List<Object>}\ \},\ \{\ \texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{AbstractList<Object>}\ \},\ \{\ \texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{Vector<Object>}\ \}\ \}$$

This means that this type unification has three unifiers as its results.

4. TYPE INFERENCE

The language for our type inference algorithm is given in figure 3. It is an abstract representation of a core of Java 5.0. The input of the type inference algorithm is a set of abstract syntax trees representing Java 5.0 classes, where the parameters, return types, and local variables of the methods are not necessarily type annotated (underlined in figure 3). The type inference algorithm infers the absent type annotations. The result of the algorithm contains for each method an intersection of function types and the corresponding typings for the local variables. The intersection of function types of the methods describes the different possible typings of its parameters and its return types.

4.1 The Algorithm

The basic idea of the algorithm is that each expression, each statement and each block is typed by simple types and that each method is typed by function types. These types are determined step by step during the run of the algorithm.

Type assumptions: First, we assume for each expression, for each statement, and for each block a type variable as a type–placeholder. The types of the methods are assumed as function types, which consists also of type–placeholders for each argument and the return type.

Run over the abstract syntax tree: During a run over the abstract syntax tree, the types of each expression, each statement, and each block are determined. This is done step by step. At each position of the abstract syntax tree there are type assumptions of the expressions, the statements, and the blocks, respectively, and there are conditions for these types given by the Java 5.0 type system. The type assumptions are unified by type unification as the respective type–conditions define.

For example if we determine the type of Assign($a, expr$) ($a = expr$), we have type assumptions ty_a and ty_{expr} for a respective $expr$. The type–condition for Assign defines, that it holds $ty_{expr} \leq^* ty_a$. This means that the type unification algorithm is applied to $\{\ ty_{expr} \lessdot ty_a\ \}$. After that the resulting unifiers are applied to the respective type assumptions.

There are rules for each Java 5.0 construct, which define the type–conditions for the corresponding types.

Multiplying the assumptions: In two cases the set of type assumptions is multiplied:

- If the result of the type unification contains more than one unifier, for each unifier a new set of type assumptions is generated.
- If during a method call there are different receivers, which can invoke the method, for each receiver a new set of type assumptions is generated.

In both cases the algorithm is continued on both sets of type assumptions.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

New method type parameters: If at the end, there are type–placeholders contained in type assumptions, these type–placeholders are replaced by new introduced method type parameters.

Intersection types: If at the end, there is more than one set of type assumptions for a method, this method has an intersection type, which is then generated.

4.2 Type Inference Example

We consider again the *matrices* example from the introduction. We apply the algorithm to the corresponding abstract syntax tree of the class `Matrix` (fig. 1), where the underlined type annotations are erased. In the first step **type assumptions** all expressions, statements, and the block are typed by type–placeholders. In the following we consider some steps of the **run over the abstract syntax tree**.

First, the New–statement New(`Matrix`, ()) (in concrete syntax: `new Matrix();`) gets the type `Matrix` as its type assumption.

Then, the statement Assign(`ret`, New(`Matrix`, ())) (`ret = new Matrix();`) should be typed. For this the type assumption of `ret` is also necessary. The type assumption of `ret` is the type–placeholder β. The type–condition for Assign–statements, leads to the condition `Matrix` $\lessdot \beta$. The type unification gives two unifiers: $\{ \beta \mapsto$ `Matrix` $\}$ and $\{ \beta \mapsto$ `Vector<Vector<Integer>>` $\}$. This means that the algorithm's step, **multiplying the assumptions**, is applied and we get two sets of type assumptions. In the first one the type of `ret` is assumed as `Matrix` and in the second one it is assumed as `Vector<Vector<Integer>>`.

Next, we consider the type calculation of the statement MethodCall(MethodCall(`m`, `elementAt`, `k`), `elementAt`, `j`) (`m.elementAt(k).elementAt(j)`) in the innermost while–loop. First, the type of MethodCall(`m`, `elementAt`, `k`) is determined. The type assumption of `m` is the type–placeholder α. Now all types are considered, which can invoke a method `elementAt`. In our context it is `Vector<T>` with `elementAt` : `Vector<T>` $\to$ `T`. The type–condition of the methodcall–rule defines that for a new type–placeholder δ, it holds $\alpha \lessdot$ `Vector<`δ`>`. There are two unifiers: $\{ \alpha \mapsto$ `Vector<`δ`>` $\}$ and $\{ \alpha \mapsto$ `Matrix`, $\delta \mapsto$ `Vector<Integer>` $\}$. This means that we get, by the algorithm's step **multiplying the assumptions**, two different type assumptions δ and `Vector<Integer>` for MethodCall(`m`, `elementAt`, `k`). This expression is simultaneously the receiver of the second method call. This means that on the one hand for δ all types are considered, which can invoke a method `elementAt`. This is again `Vector<T>`. On the other hand it is determined wether `Vector<Integer>` can invoke `elementAt`. This is also possible. This means following the type–condition of the methodcall–rule, that for a new type–placeholder ϵ it must hold $\delta \lessdot$ `Vector<`ϵ`>` and for a further type–placeholder ϵ' it must hold `Vector<Integer>` $\lessdot$ `Vector<`ϵ'`>`. The first unification again gives the unifiers $\{ \delta \mapsto$ `Vector<`ϵ`>` $\}$ and $\{ \delta \mapsto$ `Matrix`, $\epsilon \mapsto$ `Vector<Integer>` $\}$. The second unification gives $\{ \epsilon' \mapsto$ `Integer` $\}$. This means, that we get for MethodCall(MethodCall(`m`, `elementAt`, `k`), `elementAt`, `j`) =: **(*)** three type assumptions `Integer`, ϵ, and `Vector<Integer>` and for the parameter `m` we get the type assumptions `Vector<Vector<`ϵ`>>`, `Vector<Matrix>`, and `Matrix`.

Then the type for Add(... , **(*)**) is determined. The type–condition for the addition demands that its arguments are subtypes of `Integer`. The means, that it must holds `Integer` $\lessdot$ `Integer`, $\epsilon \lessdot$ `Integer`, and `Vector<Integer>` $\lessdot$ `Integer`. It is obvious, that the last unification fails. This means that the algorithm's step **erase type assumptions** is applied and the corresponding set of type assumptions is erased. From this, it follows, that for `m` there remains two adapted type assumptions `Vector<Vector<Integer>>` and `Matrix`.

During the rest of the running nothing happens to the type assumptions of the parameter `m`.

The statement Return(`ret`) (`return ret;`) determines the return type of this presently considered method `mul`. As the type assumptions of the local variable `ret` are `Matrix` and `Vector<Vector<Integer>>`, these two are the type assumptions for the return type.

In the last step of the algorithm, **intersection types**, this leads to the following inferred type for `mul`:

$$\begin{array}{lll} \texttt{mul} & : & \texttt{Vector<Vector<Integer>>} \to \texttt{Matrix} \\ & \& & \texttt{Matrix} \to \texttt{Matrix} \\ & \& & \texttt{Vector<Vector<Integer>>} \\ & & \qquad \mapsto \texttt{Vector<Vector<Integer>>} \\ & \& & \texttt{Matrix} \mapsto \texttt{Vector<Vector<Integer>>} \quad . \end{array}$$

This is the result which we expected in the first section.

4.3 Principle Type Property

First, we will give a definition of a principle Java 5.0 type. The definition is a generalization of the corresponding definition in functional programming languages [3].

Definition 5. An intersection type of a method `m` in a class `C`

$$\begin{array}{ll} m: & (\theta_{1,1} \times \ldots \times \theta_{1,n} \to \theta_1) \\ & \& \ldots \& \\ & (\theta_{m,1} \times \ldots \times \theta_{m,n}, \to \theta_m) \end{array}$$

is called *principle* if for any type annotated method declaration

$$\textit{rty}\ \texttt{m(}\textit{ty1}\ \texttt{a1 , ... ,}\ \textit{tyn}\ \texttt{an) \{ ... \}}$$

there is an element $(\theta_{i,1} \times \ldots \times \theta_{i,n}, \to \theta_i)$ of the intersection type and there is a substitution σ, such that

$$\sigma(\theta_i) = \textit{rty}, \sigma(\theta_{i,1}) = \textit{ty1}, \ldots, \sigma(\theta_{i,n}) = \textit{tyn}$$

THEOREM 1. *If we consider only simple types with unbounded type variables and without wildcards, the type inference algorithm calculates a principle type.*

4.4 Resolving Intersection Types

In conventional Java no intersection types for methods are allowed. This means that the compiler cannot translate them. Therefore, we need an approach to deal with intersection types after they are inferred. There are three possibilities.

The first one is to present the user with all inferred types and the user has to select one of them. Subsequently code is generated for that type. At the moment we have implemented this approach.

Another approach would be to generate code for each element of the intersection type. This approach would have the advantage, that later on all inferred types for the method would be usable. The disadvantage is, that the same code appears several times in the byte–code file.

The third idea is to generate the code for each method only once. However, for each type of the intersection type an entry in the constant–pool is built. This means that the same executable code is referenced by different entries in the constant–pool. For this approach we have to do further investigations.

5. IMPLEMENTATION

In order to present a proof of concept, we have integrated the type inference system in a Java compiler. The compiler itself has been implemented in Java by using the tools JLex [2] and jay [12].

In its analysis phase the compiler parses a Java program and creates an *Abstract Syntax Tree* (subsequently called *AST*). This AST forms, together with some other basic data structures, the input data for the *Type Inference Algorithm* (subsequently called *TIA*) described in section 4. The TIA calculates the missing type annotations and performs a general type checking. In doing so, it replaces the common semantic check.

5.1 Basic Data Structures

5.1.1 TypePlaceholder

All the types of the Java program to be compiled are represented in the AST by instances of subclasses of the abstract class *Type*. In order to allow programmers to omit type annotations for method declarations and local variable declarations, we have to extend the type hierarchy by introducing another subclass of *Type*.
This subclass is an auxiliary data structure for the TIA. Its instances act as placeholders for the individual declaration types. The class is therefore called *TypePlaceholder*.

5.1.2 TypeAssumption

The implementation of the type assumptions described in section 4 is realized by the abstract class *TypeAssumption*. This class is, besides the AST, the main data structure for the TIA. It basically maps an identifier onto an assumed type by storing a *String* instance and a *Type* instance.

At the beginning of the TIA an initial set of *TypeAssumption* instances is created for all field declarations (field variables and methods). During the processing of the TIA when more and more knowledge about the types is gained, this set is extended by adding new *TypeAssumption* instances or by modifying old ones.

5.1.3 Substitution

While *TypeAssumption* is the implementation for mapping an identifier onto a type, the class *Substitution* maps a type placeholder onto a calculated type. A *Substitution* instance stores a *TypePlaceholder* reference and the corresponding *Type* instance which the placeholder will be replaced with. Normally for each unifier provided by the unification algorithm (see section 3) a *Substitution* instance is created.

Through the method *Substitution.execute()* the type replacement for this particular placeholder in the AST can be invoked (see section 5.2).

5.2 Substitution Based Approach

The TIA theoretically described in section 4 follows an approach which is mainly based on type assumptions. The algorithm cyclically collects type information and unification results in order to extend and specify existing sets of type assumptions.

Type substitutions however play a minor role in this approach and are only used as an auxiliary means. The unifiers provided by the unification algorithm are usually discarded after their type substitution has been applied on the sets of type assumptions.

The TIA's output data structure consists of multiple sets of type assumptions which represent a theoretical image of the Java program's type configuration. Type unifiers or type substitutions are not part of the output data structure.

This assumption based approach is very difficult to implement as such a set of type assumptions as a whole cannot be applied to the AST. The type information must be broken down into small, executable instructions. These instructions are identical to the type substitutions, though. For the implementation it is crucial to add all type substitutions as *Substitution* instances to the TIA's output data!

The implementation still uses, according to the TIA's specification, type assumptions for calculating types, but in terms of modifying the AST, type substitutions are more important.

Therefore the implemented TIA returns a data structure consisting of multiple tuples of *TypeAssumption* sets and *Substitution* sets. Each tuple represents a possible type configuration for the Java program.

5.3 Applying the Output Data

The question facing our implementation is, how to apply such a set of type substitutions returned by the TIA to the AST.

The most obvious solution would be to use the set of type substitutions as input data for a second algorithm which is responsible for applying them to the AST. Considering that the whole AST would have to be traversed again, the performance of this solution would not be very good.

Therefore we choose a totally different approach which is based on the *Observer Design Pattern* [7]. According to this design pattern, many *observers* (also called *listeners*) register themselves at an object they are interested in, so that they can be notified about its state changes.

In our case the observers are all the AST components which store a *TypePlaceholder* object. Such a component registers itself as an observer at the *TypePlaceholder* whose state changes it is interested in. The state changes are the type replacements and substitutions respectively. Such an observer is represented by the interface *ITypeReplacementListener* and is notified by a method call to its interface method *replaceType(ReplaceTypeEvent e)*. The observer can then replace its *TypePlaceholder* with the new type it receives via the *ReplaceTypeEvent*.

Each *TypePlaceholder* stores its *ITypeReplacementListeners* in a *Vector* called *m_ReplacementListeners* and notifies all registered observers when the method *fireReplaceTypeEvent()* is called (see figure 4).

As all observers are stored in a field variable of *TypePlaceholder*, it is important that all observers who are interested in a type placeholder A register at the same *TypePlaceholder* instance representing A. This means that within the AST there must not be more than one instance for the type placeholder A.

In order to achieve this, we prohibit the creation of *TypePlaceholder* objects by defining its constructor private. Instead we provide the static *Factory Method* [7] *TypePlaceholder.fresh()* which creates a new *TypePlaceholder* object and stores it in a central registry. An existing *TypePlaceholder* can be retrieved from the registry by the static method *TypePlaceholder.getInstance(String name)*.

So the following happens after the TIA has finished. The

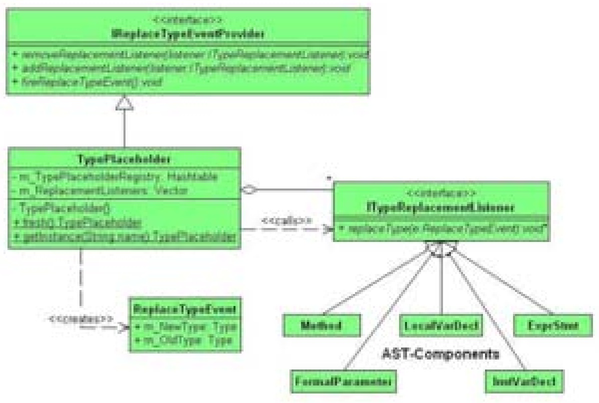

<<interface>>
IReplaceTypeEventProvider
+ removeReplacementListener(listener: ITypeReplacementListener): void
+ addReplacementListener(listener: ITypeReplacementListener): void
+ fireReplaceTypeEvent(): void
TypePlaceholder
- m_TypePlaceholderRegistry: Hashtable
- m_ReplacementListeners: Vector
- TypePlaceholder()
+ fresh(): TypePlaceholder
+ getInstance(String name): TypePlaceholder
*
<<calls>>
<<interface>>
ITypeReplacementListener
+ replaceType(e: ReplaceTypeEvent): void
<<creates>>
ReplaceTypeEvent
+ m_NewType: Type
+ m_OldType: Type
Method
LocalVarDecl
ExprStmt
AST-Components
FormalParameter
InstVarDecl

Figure 4: Implemented Observer Pattern

programmer has to choose between the possible type configuration returned by the TIA. On each *Substitution* object of the selected configurations the method *execute()* is called. This method requests the unique *TypePlaceholder* instance of its type placeholder from the registry and passes its calculated type to *TypePlaceholder.replaceWith(Type newType)* which triggers *TypePlaceholder.fireReplaceTypeEvent()*. Subsequently all registered *ITypeReplacementListeners* are notified and replace their *TypePlaceholder* with the calculated type.

6. CONCLUSION AND OUTLOOK

We gave a type inference algorithm for Java 5.0. The algorithm calculates a method's parameter types and its return type. For type terms without bounded type variables and without wildcards a principle type is inferred. The type inference algorithm is based on the type unification algorithm. It is possible to consider the type inference algorithm as a generic algorithm parameterized by the type unification algorithm. At the moment we gave a type unification algorithm, which calculates unifiers for type terms without bounded type variables and without wildcards.

For the introduction of bounded type variables step 4 of the type unification algorithm (section 3) must be extended. The pairs of the form $a \lessdot ty1$ and $a \lessdot ty2$, where the types $ty1$ and $ty2$ are not unifiable, should be transformed to $a \lessdot ty1\&ty2$. Furthermore we aim to discover wether such a strategy leads to a principle type or not.

The introduction of wildcards leads to a problem if the type unification is computable. This is marked as open in [18]. There is an idea to solve this problem by extending our type unification algorithm.

Furthermore, we are working at a translation function for the byte–code to integrate intersection types as discussed in section 4.4.

7. REFERENCES

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] E. Berk. *JLex: A lexical analyzer generator for Java(TM)*. http://www.cs.princeton.edu/ appel/modern/java/JLex, 1.2 edition, October 1997.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[4] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 15–34, New York, NY, USA, 2004. ACM Press.

[5] J. Eifrig, S. Smith, and V. Trifonov. Type Inference for Recursively Constrained Types and its Application to Object Oriented Programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[6] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005.

[7] Gang of Four. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] A. Goldberg and D. Robson. *Smalltalk–80: The Language and Its Implementation.* Addison-Wesley, 1983.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification.* The Java series. Addison-Wesley, 3rd edition, 2005.

[10] S. P. Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[11] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[12] B. Kühl and A.-T. Schreiner. jay – Compiler bauen mit Yacc und Java. *iX*, 1999. (in german).

[13] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[14] R. Milner. *The definition of Standard ML (Revised).* MIT Press, Cambridge, Mass., 1997.

[15] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming*, LNCS 615:329–349, July 1992.

[16] M. Plümicke. OBJ–P *The Polymorphic Extension of* OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4, 1999.

[17] M. Plümicke. Type unification in Generic–Java. In M. Kohlhase, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, July 2004.

[18] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.

Infinite Streams in Java

Dominik Gruntz
Institute for Mobile and Distributed Systems
University of Applied Sciences, Northwestern Switzerland
Steinackerstrasse 5, CH-5210 Windisch, Switzerland
dominik.gruntz@fhnw.ch

ABSTRACT

Programming languages which support lazy evaluation allow the definition of infinite streams as for example the stream of natural numbers or of prime numbers. Such streams are infinite in the sense that arbitrary many elements can be accessed as these elements are computed “on demand”.

This paper describes how infinite streams can be implemented in Java, a language which does not support lazy evaluation directly. Two possible implementations are described and compared. Furthermore it is shown how streams can be defined as fixed points of maps on infinite streams and how formal power series can be defined using infinite streams. As user interface to work with such streams Groovy is used.

1. INTRODUCTION

Infinite streams are streams which conceptionally contain infinitely many objects. Such streams can be represented using lazy evaluation, a strategy where expressions are only evaluated when needed. Infinite streams can for example be used to represent the stream of all prime numbers or the stream of the coefficients of a power series. Infinite streams are useful in applications where the number of elements which need to be accessed is not known in advance.

Lazy evaluation became popular in the context of functional languages [1]. Examples of functional languages which support lazy evaluation as default evaluation strategy are Haskell and Miranda. Some languages provide library functions which allow to emulate lazy evaluation (e.g. Scheme defines the two functions `delay` and `force` in its standard library).

The Java language does not support lazy evaluation directly, but it can be simulated with a functor object definition (as e.g. provided by the Jakarta Commons Functor library [2] or by JGA [3]). Such a functor is usually defined by a method signature in an interface and is implemented in an inner class. Implementations of infinite streams for Java have been proposed in [4, 5].

In Sections 2 and 3 we present two implementations to represent infinite streams in Java. These implementations correspond to the linked list and array list implementations of finite lists provided by the Java collection framework. Existing stream implementations for Java follow the linked list model.

Infinite streams are often recursively defined, i.e. the lazily evaluated part of the stream contains a reference to the defined stream itself. This property can be used to define infinite streams as fixed points of functions on infinite streams. In Section 4 we show, how fixed point definitions can be implemented in Java and which properties a stream implementation must meet so that it can be used in such definitions.

We use this technology to define infinite power series in Section 5 and close the article with Section 6 demonstrating how Groovy can be used as user interface.

PPPJ 2006, August 30 – September 1, Mannheim, Germany
Copyright 2006 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. LINKED STREAMS

We start the implementation with the definition of the interface `Stream` to represent immutable infinite streams. This interface contains nested interfaces for selectors and functors which are used to define filters and maps on infinite streams.

```
public interface Stream<E> {

   interface Selector<T> { boolean select(T x); }
   interface UnaryFunctor<T,R> { R eval(T x); }
   interface BinaryFunctor<T1,T2,R> {
      R eval(T1 x, T2 y);
   }

   E getHead();
   E get(int index);
   Stream<E> getTail();

   java.util.Iterator<E> iterator();
   String toString(int order);

   // add an element in front
   Stream<E> prepend(E x);

   // select elements
   Stream<E> select(Selector<? super E> s);

   // map a function onto the stream
   <R> Stream<R> map(UnaryFunctor<? super E, R> f);

   // combine two streams
   <T,R> Stream<R> zip(
      BinaryFunctor<? super E, ? super T, R> f,
      Stream<T> stream);
}
```

The first implementation follows the idea of linked lists, where the evaluation of the remainder is delayed until it is accessed. For that purpose we define an interface `LazyTail` whose method `eval` returns another (lazily evaluated) infinite stream when evaluated. This implementation corresponds to the implementations presented in [4, 5]. To keep the code simple we only show the necessary methods and discuss further methods below.

```
public class LinkedStream<E> implements Stream<E> {

   public interface LazyTail<E> {
      LinkedStream<E> eval();
   }

   private E head;
   private LinkedStream<E> tail; //assigned on demand
   private LazyTail<E> lazyTail; //delayed tail rule
   public LinkedStream(E head, LazyTail<E> tail) {
      this.head = head;
      this.lazyTail = tail;
   }

   public E getHead(){ return head; }
   public LinkedStream<E> getTail(){
      if(tail == null) {
         tail = lazyTail.eval(); lazyTail = null;
      }
      return tail;
   }
   ...
}
```

Method `getHead` returns the head element of the stream, and `getTail` computes and returns the tail of the stream. Since the remainder of a stream should only be evaluated once, we store the result in the field `tail` and free the reference to the computation procedure so that the garbage collector can reclaim it.

As an example we define the infinite stream of integers:

```
public class StreamTest {
   static LinkedStream<Integer> integersFrom(
                  final int start) {
      return new LinkedStream<Integer>(start,
         new LinkedStream.LazyTail<Integer>(){
            public LinkedStream<Integer> eval(){
               return integersFrom(start+1);
            }
         }
      );
   }

   public static void main(String[] args) {
      Stream<Integer> integers = integersFrom(0);

      Stream<Integer> s = integers;
      for(int i=0; i<20; i++){
         System.out.print(s.getHead() + " ");
         s = s.getTail();
      }
      System.out.println("...");
   }
}
```

When this program is executed the following result is printed on the console:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
```

The `get` method allows accessing an element at a specified position. To do that, the stream is traversed and, if necessary, built up to the specified position. This method could also be defined in an abstract base class available for all stream implementations as it does not use implementation specific aspects.

```
public E get(int index) {
   if (index < 0) throw
      new IndexOutOfBoundsException("negative index");
   Stream<E> stream = this;
   while (index-- > 0) stream = stream.getTail();
   return stream.getHead();
}
```

For the `map` method a new stream is generated where the given function is mapped on each element. The implementation defines an inner class for the lazy tail. Fields accessed in the outer scope are copied and have to be declared final. The `zip` and `select` functions are defined similarly.

```
public <R> LinkedStream<R> map(
              final UnaryFunctor<? super E, R> f) {
   return new LinkedStream<R>(
      f.eval(getHead()),
      new LazyTail<R>(){
         public LinkedStream<R> eval(){
            return getTail().map(f);
         }
      }
   );
}
```

The `prepend` method adds a new head element. This method is problematic as it is strict on the stream it is called. In particular, the head element of the stream to which a new element is prepended is evaluated as only the tail of the series is lazily defined.

```
public Stream<E> prepend(E head){
   return new LinkedStream<E>(
      head,
      new LazyTail<E>(){
         public LinkedStream<E> eval(){
            return LinkedStream.this;
         }
      }
   );
}
```

The evaluation of a stream s to which a new head element e is added could be avoided, if the construct

```
new LinkedStream<E>(e, new LazyTail<E>(){
   public LinkedStream<E> eval(){ return s; }
})
```

were used.

We close this section with the definition of the streams of integers, squares and fibonacci numbers. Both methods `integersFrom` and `fibonacci` are recursive and contain calls to itself in the eval method defined in the anonymous class.

```
public class StreamTest2 {

   public static LinkedStream<Integer> integersFrom(
                  final int start) {
      return new LinkedStream<Integer>(
         start,
         new LinkedStream.LazyTail<Integer>(){
            public LinkedStream<Integer> eval(){
               return integersFrom(start+1);
            }
         }
      );
   }
```

```
    private static LinkedStream<Integer> fibonacci(
                    final int a, final int b){
       return new LinkedStream<Integer>(
          a,
          new LinkedStream.LazyTail<Integer> () {
             public LinkedStream<Integer> eval(){
                return fibonacci(b, a+b);
             }
          }
       );
    }

    public static void main(String[] args) {
       Stream<Integer> integers = integersFrom(0);
       System.out.println(integers);

       Stream<Integer> squares = integers.map(
          new Stream.UnaryFunctor<Integer, Integer>(){
             public Integer eval(Integer x){
                return x*x;
             }
          }
       );
       System.out.println(squares);

       Stream fibonaccy = fibonacci(0, 1);
       System.out.println(fibonaccy);
    }
 }
```

This program generates the following output:

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ...
 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 ...
 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...
```

3. ARRAY STREAMS

In class LinkedStream an infinite stream is represented as a linked list. Accessing the n^{th} element requires $O(n)$ operations. An alternative would be to store the stream elements in an array or a hash table where they can be accessed in constant time. This is how array lists are implemented in the Java collection framework. This strategy can also be applied to infinite streams. Instead of a functor for the computation of the remainder of a stream a functor for the computation of the n^{th} term of the stream is provided.

We present a simple implementation of this approach. Class ArrayStream contains only a minimal set of methods; additional methods of the Stream interface are discussed below.

```
 import java.util.HashMap;
 public class ArrayStream<E> implements Stream<E> {

    public interface Coefficients<E> {
       E get(int index);
    }

    private Coefficients<E> lazyCoeff;
    private HashMap<Integer,E> coeff =
                       new HashMap<Integer,E>();

    public ArrayStream(Coefficients<E> lazyCoeff){
       if(lazyCoeff == null) throw
          new IllegalArgumentException();
       this.lazyCoeff = lazyCoeff;
    }

    public E get(int index){
       if (index < 0) yhrow new
          IndexOutOfBoundsException("negative index");
       Integer n = index;
       if(!coeff.containsKey(n))
          coeff.put(n, lazyCoeff.get(index));
       return coeff.get(n);
    }

    public E getHead(){ return get(0); }

    public ArrayStream<E> getTail(){
       return new ArrayStream<E>(
          new Coefficients<E>(){
             public E get(int index){
                return ArrayStream.this.get(index+1);
             }
          }
       );
    }
 }
```

Method get first checks whether the requested term of the stream has already be computed. If not, it is computed with the provided coefficient method. getHead simply returns the first term, and getTail constructs a new ArrayStream which returns the terms shifted by one. Note, that the terms of the tail stream are stored in that stream's coefficient map as well. This waste of storage could be avoided (at the cost of an additional method call) with a *no-cache* flag added to class ArrayStream.

We present the implementation of two additional methods of the Stream interface. The prepend method defines a new coefficient method which returns the new head element and the terms of the original stream shifted by one. In contrast to the prepend method of LinkedStream the evaluation of the head element of the this stream is not forced.

```
 public Stream<E> prepend(final E head){
    return new ArrayStream<E>(
       new Coefficients<E>(){
          public E get(int index){
             if(index == 0) return head;
             else return ArrayStream.this.get(index-1);
          }
       }
    );
 }
```

The select function is more tricky for array streams than for linked streams. If the k^{th} element is accessed, the elements with index 0 up to $k-1$ already have to be known. This is ensured in the implementation below by accessing the directly preceding element which is either taken out of the hash map or computed. Moreover, the coefficient procedure has to maintain an index into the original stream.

```
 public Stream<E> select(final Selector<? super E> s){
    final ArrayStream<E> stream = new ArrayStream<E>();
    stream.lazyCoeff = new Coefficients<E>(){
       int pos = 0;
       public E get(int index){
          if(index > 0) stream.get(index-1);
          E x = null;
          do {x = ArrayStream.this.get(pos++);}
          while(!s.select(x));
          return x;
       }
    };
    return stream;
 }
```

Unfortunately, the use of select operations leads to deeper recursion levels with this stream representation and may end earlier in a StackOverflowError. Moreover, random access is not used in typical stream applications.

4. FIXED POINT DEFINITIONS

A stream often depends on itself as e.g. the stream of integers or fibonacci numbers shown above. This self recursion is revealed in the lazily evaluated recursive calls of the methods `integersFrom` or `fibonacci`. This recursion property can also be made more visible and be used to define a stream as a fixed point of a map on the domain of streams [6].

Such maps are defined with method `map` of the interface `StreamMap` (defined in the interface `Stream`)

```
interface StreamMap<T> {
   Stream<T> map(Stream<T> stream);
}
```

and new streams are constructed with the static generic method `fixedpoint`

```
static <T> Stream<T> fixedPoint(StreamMap<T> map){..}
```

which constructs a stream which is a fixed point of the given map.

In particular, we think of maps which do not perform operations on their argument but rather simply include it in a new stream which is returned as result. In particular, the head element of the defined stream must not depend on the argument. This way bootstrapping of the construction of the stream is possible.

As an example we show the definition of the stream of ones with a fixed point. This stream is the fixed point of a map which prepends the element one to its argument.

```
Stream<Integer> ones = fixedPoint(
   new StreamMap<Integer>(){
      public Stream<Integer> map(Stream<Integer> s){
         return s.prepend(1);
      }
   }
);
```

We now discuss how the `fixedPoint` method is implemented for `LinkedStream`s and `ArrayStream`s. In both cases a new stream is constructed and passed to the given map. The result is another stream of the same type. The fields of this stream are copied to the initially generated stream. This way the fixed point of the map is constructed. The resulting code looks similar for both classes.

```
public class LinkedStream<E> implements Stream<E> {
   private LinkedStream(){}
   public static <T> Stream<T> fixedPoint(
                                StreamMap<T> map){
      LinkedStream<T> s1 = new LinkedStream<T>();
      LinkedStream<T> s2 = (LinkedStream<T>)map.map(s1);
      s1.head = s2.head;
      s1.lazyTail = s2.lazyTail;
      return s2;
   } ...
}

public class ArrayStream<E> implements Stream<E> {
   private ArrayStream(){}
   public static <T> Stream<T> fixedPoint(
                                StreamMap<T> map){
      ArrayStream<T> s1 = new ArrayStream<T>();
      ArrayStream<T> s2 = (ArrayStream<T>)map.map(s1);
      s1.lazyCoeff = s2.lazyCoeff;
      return s2;
   } ...
}
```

Let us now define some streams as fixed points of stream maps. The first example is the periodic sequence [A, B, C, A, B, C, A, B, C, ...]. This stream is simply the fixed point of a map which pretends the strings “A”, “B” and “C” to its argument. The statement

```
System.out.println(
   fixedPoint(
      new StreamMap<String>(){
         public Stream<String> map(Stream<String> s){
            return s.prepend("C").prepend("B").
                                       prepend("A");
         }
      }
   )
);
```

generates the output

```
A B C A B C A B C A B C A B C ...
```

Next we define the stream of integers as fixed point of a map which takes a stream, adds one to each element and prepends zero.

```
integers = fixedPoint(
   new StreamMap<Integer>(){
      public Stream<Integer> map(Stream<Integer> s){
         return s.map(
            new UnaryFunctor<Integer, Integer>(){
               public Integer eval(Integer x){
                  return x+1;
               }
            }
         ).prepend(0);
      }
   }
);
```

This example only works if the `prepend` method does not evaluate the stream to which a new head element is prepended. Otherwise the call `s.map` is evaluated which results in a `NullPointerException` as the head element of the stream to be constructed by this fixed point definition is not yet defined. As a consequence, this definition only works with the fixed point method of class `ArrayStream`.

Fibonacci numbers can be expressed as the fixed point of the map

$$fibs \to 0 : (+ \;\; fibs \;\; (1 : fibs))$$

which leads to the following definition:

```
class Add implements
   Stream.BinaryFunctor<Integer, Integer, Integer> {
   public Integer eval(Integer x, Integer y){
      return x+y;
   }
};

Stream<Integer> fibonacci = fixedPoint(
   new StreamMap<Integer>() {
      public Stream<Integer> map(Stream<Integer> f){
         return f.zip(new Add(),
            f.prepend(1)).prepend(0);
      }
   }
);
```

Again, this definition only works with class `ArrayStream`. In this example the `zip` function accesses the head element of the f stream in order to compute the head element of the resulting stream.

With the alternative map

$$\mathit{fibs} \to (+ \;\; (0 : \mathit{fibs}) \;\; (0 : (1 : \mathit{fibs})))$$

the fixed point can be computed with both implementations. The two head elements which are accessed in the zip function within class LinkedStream are defined.

```
Stream<Integer> fibonacci = fixedPoint(
   new StreamMap<Integer>() {
      public Stream<Integer> map(Stream<Integer> f){
         return f.prepend(0).zip(new Add(),
            f.prepend(1).prepend(0));
      }
   }
);
```

If we print out the stream fibonacci we receive the following result:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...
```

One advantage of fixed point definitions is that the recursively referenced data structure is not reevaluated (in contrast to the definition with methods which are recursively called in the lazily evaluated part as shown in sections 2 and 3). This leads to more efficient programs than those where a stream is reconstructed by a method call [7].

5. INFINITE POWER SERIES

In order to apply the streams derived in this article we define infinite power series, i.e. power series where any number of terms can be asked. This is useful because the number of terms required is often not known in advance.

Infinite power series use a stream to store their coefficients. It turns out, that many series operations have extremely simple implementations when using lazy evaluation and fixed point definitions.

The interface of class PowerSeries is shown below. The rational coefficients are implemented in class Rational. With the methods diff and integrate a series can be differentiated and integrated. The integration constant can be provided as an optional parameter. The shift method allows to multiply a power series with the factor x (and to add a constant term).

```
public interface PowerSeries {

   public Rational coeff(int index);

   public PowerSeries add(PowerSeries value);
   public PowerSeries subtract(PowerSeries value);
   public PowerSeries negate();
   public PowerSeries multiply(Rational value);
   public PowerSeries multiply(PowerSeries value);
   public PowerSeries divide(PowerSeries value);

   public PowerSeries diff();
   public PowerSeries integrate();
   public PowerSeries integrate(Rational c);
   public PowerSeries shift(Rational c);
}
```

In the implementing class the coefficients are stored in the field coefficients of type ArrayStream<Rational>. As an example we show the method integrate which integrates a power series term by term.

```
public PowerSeries integrate(final Rational c){
   return new PowerSeries(
      new ArrayStream<Rational>(
         new ArrayStream.Coefficients<Rational>(){
            public Rational get(int k){
               if(k==0) return c;
               else return coeff(k-1).divide(
                  new Rational(k)
               );
            }
         }
      )
   );
}
```

Moreover, the power series class also defines a fixedpoint method which allows to define power series as fixed points of a power series map.

```
interface PowerSeriesMap {
   PowerSeries map(PowerSeries series);
}

public PowerSeries fixedpoint(PowerSeriesMap map){
   PowerSeries s1 = new PowerSeries();
   PowerSeries s2 = map.map(s1);
   s1.coefficients = s2.coefficients;
   return s2;
}
```

The fixedpoint method works as long as all methods used in the power series map do not access the coefficients field in an eagerly evaluated part. For example, the multiply method which scales a power series which a rational number must not call the map function on the coefficients but rather has to define a new coefficient function.

As an example we define the power series for $\exp(x)$. The equation

$$\exp(x) = \int \exp(x)\, dx + \exp(0)$$

can directly be used to define the exponential series at $x = 0$ using a fixed point:

```
PowerSeries exp = PowerSeries.fixedpoint(
   new PowerSeriesMap(){
      public PowerSeries map(PowerSeries exp){
         return exp.integrate(Rational.ONE);
      }
   }
);

System.out.println(exp);
=> 1 + x + 1/2 x^2 + 1/6 x^3 + 1/24 x^4 + O(x^5)
```

As another example we show the definition of the power series for $\tan(x)$ which is the fixed point of the map

$$\tan(x) \to \int \left(1 + \tan(x)^2\right)\, dx$$

which leads to the following definition in Java:

```
PowerSeries tan = fixedpoint(
   new PowerSeriesMap(){
      public PowerSeries map(PowerSeries tan){
         return tan.multiply(tan)
            .add(new PowerSeries(Rational.ONE))
            .integrate();
      }
   }
);

System.out.println(tan.toString(9));
=> x + 1/3 x^3 + 2/15 x^5 + 17/315 x^7 + O(x^9)
```

6. GROOVY

Groovy [8] is a scripting language for the Java platform whith is specified under JSR 241. Groovy provides a more natural notation for closures than Java and therefore could be used to define infinite streams. Groovy scripts can be compiled straight to Java byte-code and can be used by regular Java applications.

Since we already have infinite stream implementations in Java we use the Groovy feature that existing Java classes and libraries can be used directly in scripts. Groovy thus can be seen as an interactive shell to work with infinite streams. In order to support the definition of infinite streams with Groovy closures, we have defined a Groovy specific stream mapper classes for array-streams over integers and for power series over rationals. These classes implement the Groovy specific operator methods

```
import groovy.lang.*;
public class ArrayStream {
   private lazy.Stream<Integer> s;

   public ArrayStream(final Closure c){
      s = new lazy.ArrayStream<Integer>(
         new lazy.ArrayStream.Coefficients<Integer>(){
            public Integer get(int index){
               return (Integer)c.call(index);
            }
         }
      );
   }

   Integer getAt(int index){
      return s.get(index);
   }
}
```

Besides the methods shown above we implemented methods to map a boolean closure on a stream, to select elements based on a predicate or to add two streams.

```
groovy> import groovy.*;
groovy> integers = new ArrayStream({n -> n})
Result: 0 1 2 3 4 5 6 7 8 9 10 11 ...

groovy> squares = integers.map({n -> n*n})
Result: 0 1 4 9 16 25 36 49 64 81 100 121 ...

groovy> evens = integers.select({ x -> x%2==0 })
Result: 0 2 4 6 8 10 12 14 16 18 20 22 ...

groovy> integers+squares
Result: 0 2 6 12 20 30 42 56 72 90 110 132 ...
```

We also provided a method to define streams as fixedpoint of a map on streams expressed as a Groovy closure:

```
groovy> import groovy.*;
groovy> fibs = ArrayStream.fixedpoint(
   {n -> n.prepend(0) + n.prepend(1).prepend(0)})
Result: 0 1 1 2 3 5 8 13 21 34 55 89 ...
```

Compared with the definition in section 4 the Groovy definition above is more readable and demonstrates the usefulness of Groovy as an interactive shell.

```
groovy> import groovy.*;
groovy> one = new PowerSeries(1);
Result: 1 + O(x^5)

groovy> one.exp()
Result: 1 + x + 1/2 x^2 + 1/6 x^3 + 1/24 x^4 + O(x^5)

groovy> exp = PowerSeries.fixedpoint({e->e.inte()+1})
Result: 1 + x + 1/2 x^2 + 1/6 x^3 + 1/24 x^4 + O(x^5)

groovy> cos = PowerSeries.fixedpoint(
                              {c->c.inte().inte()+1})
Result: 1 + 1/2 x^2 + 1/24 x^4 + O(x^5)

groovy> cos[36]
Result: 1/371993326789901217467999448150835200000000
```

7. CONCLUSION

We have presented two approaches to implement infinite structures in Java which correspond to a linked list and array list implementation. The array stream approach is more efficient when stream elements have to be accessed several times, whereas the linked stream implementation is more natural for typical applications.

The implementation of the prepend method differs in the two implementatons as in one case the head element of the stream to which an element is prepended is evaluated. The exact behaviour of this method is not specified in the Java interface, and additional information has to be provided.

We have also shown how the definition of streams can be expressed using fixed points using both stream implementations. Fixed point definitions however require that the fixed point argument is never evaluated.

Since Java does not support lazy evaluation as a language feature, this feature has to be emulated. This leads to rather complicated class definitions which make use of (anonymous) inner classes and which require fields in outer scopes to be declared final. But once the basic abstractions have been implemented, working with these classes is rather elegant as we have seen in the section on infinite power series. And making these classes accessible from Groovy simplifies working with infinite streams and power series even more.

The sources presented in ths article are available at `http://www.gruntz.ch/papers/infinitestreams/`

8. REFERENCES

[1] H. Abelson and G. Sussman, Structure and Interpretation of Computer Programs, 2nd ed., MIT Press, 1996.

[2] Commons Functor: Function Objects for Java, `http://jakarta.apache.org/commons/sandbox/functor/` {accessed May 2006}

[3] JGA: Generic Algorithms for Java, `http://jga.sourceforge.net/` {accessed May 2006}

[4] D. Nguyen and S. Wong, Design Patterns for Lazy Evaluation, SIGCSE 2000, Technical Symposium on Computer Science Education, Austin, Texas.

[5] U. Schreiner, Infinite Streams in Java, `http://www.innuendo.de/documentation/papers/infiniteStreams` {accessed May 2006}

[6] W.H. Burge and S.M. Watt, Infinite Structures in Scratchpad II, EUROCAL'87, Lecture Notes in Computer Science, Vol. 378, pp. 138–148, 1989.

[7] S.M. Watt, A Fixed Point Method for Power Series Computations, ISSAC'88, Lecture Notes in Computer Science, Vol. 358, pp. 206–216, 1989.

[8] Groovy Project Home, `http://groovy.codehaus.org/` {accessed May 2006}

Interaction among Objects via Roles

Sessions and Affordances in Java

Matteo Baldoni
Dipartimento di Informatica
Università di Torino - Italy
baldoni@di.unito.it

Guido Boella
Dipartimento di Informatica
Università di Torino - Italy
guido@di.unito.it

Leendert van der Torre
University of Luxembourg
Luxembourg
leendert@vandertorre.com

ABSTRACT

In this paper we present a new vision in object oriented programming languages where the objects' attributes and operations depend on who is interacting with them. This vision is based on a new definition of the notion of role, which is inspired to the concept of affordance as developed in cognitive science. The current vision of objects considers attributes and operations as being objective and independent from the interaction. In contrast, in our model interaction with an object always passes through a role played by another object manipulating it. The advantage is that roles allow to define operations whose behavior changes depending on the role and the requirements it imposes, and to define session aware interaction, where the role maintains the state of the interaction with an object. Finally, we discuss how roles as affordances can be introduced in Java, building on our language powerJava.

1. INTRODUCTION

Object orientation is a leading paradigm in programming languages, knowledge representation, modelling and, more recently, also in databases. The basic idea is that the attributes and operations of an object should be associated with it. The interaction with the object is made via its public attributes and via its public operations. The implementation of an operation is specific of the class and can access the private state of it. This allows to fulfill the data abstraction principle: the public attributes and operations are the only possibility to manipulate an object and their implementation is not visible from the other objects manipulating it; thus, the implementation can be changed without changing the interaction capabilities of the object.

This view can be likened with the way we interact with objects in the world: the same operation of switching a device on is implemented in different manners inside different kinds of devices, depending on their functioning. The philosophy behind object orientation, however, views reality in a naive way. It rests on the assumption that the attributes and operations of objects are objective, in the sense that they are the same whatever is the object interacting with them.

This view limits sometime the usefulness of object orientation:

1. Every object can access all the public attributes and invoke all the public operations of every other object. Hence, it is not possible to distinguish which attributes and operations are visible for which classes of interacting objects.
2. The object invoking an operation (caller) of another object (callee) is not taken into account for the execution of the method associated with the operation. Hence, when an operation is invoked it has the same meaning whatever the caller's class is.
3. The values of the private and public attributes of an object are the same for all other objects interacting with it. Hence, the object has always only one state.
4. The interaction with an object is session-less since the invocation of an operation does not depend on the caller. Hence, the value of private and public attributes and, consequently, the meaning of operations cannot depend on the preceding interactions with the object.

The first three limitations hinder modularity, since it would be useful to keep distinct the core behavior of an object from the different interaction possibilities that it offers to different kinds of objects. Some programming languages offer ways to give multiple implementations of interfaces, but the dependance from the caller cannot be taken into account, unless the caller is explicitly passed as a parameter of each method. The last limitation complicates the modelling of distributed scenarios where communication follows protocols.

Programming languages like Fickle [9] address the second and third problem by means of dynamic reclassification: an object can change class dynamically, and its operations change their meaning accordingly. However, Fickle does not represent the dependence of attributes and operations from the interaction. Aspect programming focuses too on the second and third issue, while it is less clear how it addresses the other ones.

Sessions are considered with more attention in the agent oriented paradigm, which bases communication on protocols ([10, 13]). A protocol is the specification of the possible sequences of messages exchanged between two agents. Since not all sequences of messages are legal, the state of the interaction between two agents must be maintained in a session. Moreover, not all agents can interact with other ones using whatever protocol. Rather the interaction is allowed only by agents playing certain roles. However, the notion of role in multi-agents systems is rarely related with the notion of session of interaction. Moreover, it is often related with the notion of organization rather than with the notion of interaction.

Roles in object oriented programming, instead, aim at modelling the properties and behaviors of entities which evolve over time, while the interaction among objects is mostly disregarded [7, 14, 18]. Hence, they adopt the opposite perspective.

In this paper, we address the four problems above in object oriented programming languages by using a new notion of role we introduced in [4] . This is inspired to research in cognitive science, where the naive vision of objects is overcome by the so called ecological view of interaction in the environment. In this view, the properties (attributes and operations) of an object are not independent from whom is interacting with it. An object "affords" different ways of interaction to different kinds of objects.

The structure of this paper is as follows. In Section 2 we discuss the cognitive foundations of our view of objects. In Section 3 we define roles in terms of affordances. In Section 4 we show how our approach to roles impact on the design of a new object oriented programming language, powerJava. Related work and conclusion end the paper.

2. ROLES AS AFFORDANCES

The naive view of objects assignes them objective attributes and operations which are independent from the observer or other objects interacting with them. Instead, recent developments in cognitive science show that attributes and operations, called *affordances*, emerge only at the moment of the interaction and change according to what kind of object is interacting with another one.

The notion of affordance has been developed by a cognitive scientist, James Gibson, in a completely different context, the one of visual perception [11] (p. 127):

> "The affordances of the environment are what it offers the animal, what it provides or furnishes, either for good or ill. The verb to afford is found in the dictionary, but the noun affordance is not. I have made it up. I mean by it something that refers to both the environment and the animal in a way that no existing term does. It implies the complementarity of the animal and the environment... If a terrestrial surface is nearly horizontal (instead of slanted), nearly flat (instead of convex or concave), and sufficiently extended (relative to the size of the animal) and if its substance is rigid (relative to the weight of the animal), then the surface affords support... Note that the four properties listed - horizontal, flat, extended, and rigid - would be physical properties of a surface if they were measured with the scales and standard units used in physics. As an affordance of support for a species of animal, however, they have to be measured relative to the animal. They are unique for that animal. They are not just abstract physical properties.
>
> The same layout will have different affordances for different animals, of course, insofar as each animal has a different repertory of acts. Different animals will perceive different sets of affordances therefore."

Gibson refers to an ecological perspective, where animals and the environment are complementary. But the same vision can be transferred to objects. By "environment" we intend a set of objects and by "animal of a given specie" we intend another object of a given class which manipulates them. Besides physical objective properties objects have affordances when they are considered relative to an object managing them. Thus, we have that the properties which characterize an object in the environment depend on the properties of the interactant. Thus, the interaction possibilities of an object in the environment depend on the properties of the object manipulating it.

How can we use this vision to introduce new modelling concepts in object oriented programming? Different sets of affordances of an object are associated with each different way of interaction with the objects of a given class. We will call a *role type* the different sets of affordances of an object. A role type represents the interaction possibilities which depend on the class of the interactant manipulating the object: the *player* of the role. To manipulate an object the caller of a method has to specify the role in which the interaction is made. But an ecological perspective cannot be satisfied by considering only occasional interactions between objects. Rather it should also be possible to consider the continuity of the interaction for each object, i.e., the state of the interaction. In terms of a distributed scenario, a session. Thus a given role type can be instantiated, depending on a certain player of a role (which must have the required properties), and the *role instance* represents the state of the interaction with that role player.

3. ROLES AND SESSIONS

The idea behind affordances is that the interaction with an object does not happen directly with it by accessing its public attributes and invoking its public operations. Rather, the interaction with an object happens via a role: to invoke an operation, it is necessary first to be the player of a role offered by the object the operation belongs to, and second to specify in which role the method is invoked. The roles which can be played depend on the properties of the player of the role (the *requirements*), since the affordances depend on the "different repertories of acts".

Thus, a class is seen as a cluster of classes gathered around a central class. The central class represents the core state and behavior of the object. The other classes, the role types, are the containers of the operations specific of the interaction with a given class, and of the attributes characterizing the state of the interaction. Not only the kind of attributes and methods depend on the class of the interacting object and on the role in which it is interacting, but also the values of these attributes may vary according to a specific interactant: they are memorized in a role instance. A role instance, thus, models the session of the interaction between two objects and can be used for defining protocols.

Since a role represents the possibilities offered by an object to interact with it, the methods of a role must be able to affect the core state of the objects they are roles of and to access their operations; otherwise, no effect could be made by the player of the role on the object the role belongs to. So a role, even if it can be modelled as an object, is, instead different: a role depends both on its player and on the object the role belongs to and it can access the state of the object the role belongs to.

Many objects can play the same role as well as the same object can play different roles. In Figure 1 we depict the different possibil-

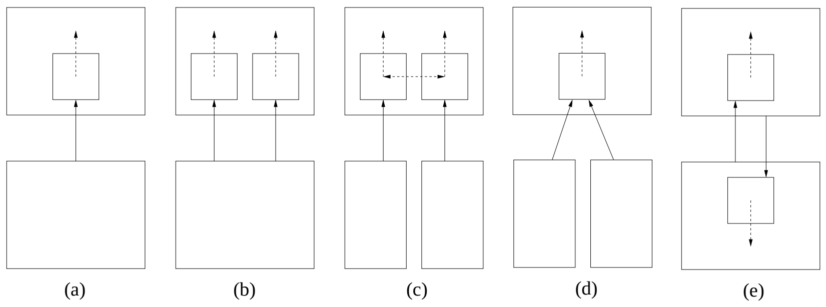


Figure 1: The possible uses of roles as affordances.

ities. *Boxes* represent objects and role instances (included in external boxes). *Arrows* represent the relations between players and their roles, *dashed arrows* the access relation between objects.

- Drawing (a) illustrates the situation where an object interacts with another one by means of the role offered by it.
- Drawing (b) illustrates an object interacting in two different roles with another one. This situation is used when an object implements two different interfaces for interacting with it, which have methods with the same signature but with different meanings. In our model the methods of the interfaces are implemented in the roles offered by the object to interact with it. Moreover, the two role instances represent the two different states of the two interactions between the two objects.
- Drawing (c) illustrates the case of two objects which interact with each other by means of the two roles of another object (the two role instances may be of the same class). This object can be considered as the context of interaction. This achieves the separation of concerns between the core behavior of an object and the interaction possibilities in a given context. The meaning of this scenario for coordination has been discussed in [5]; in this paper, we used as a running example the well-known philosophers scenario. The institution is the table, at which philosophers are sitting and coordinate to take the chopsticks and eat since they can access the state of each other. The coordinated objects are the players of the role `chopstick` and `philosopher`. The former role is played by objects which produce information, the latter by objects which consume them. None of the players contains the code necessary to coordinate with the others, which is supplied by the roles.
- In drawing (d) a degenerated but still useful situation is depicted: a role does not represent the individual state of the interaction with an object, but the collective state of the interaction of two (or more) objects playing the same role instance. This scenario is useful when it is not necessary to have a session for each interaction.
- In drawing (e) two objects interact with each other, each one playing a role offered by the other. This is often the case of interaction protocols: e.g., an object can play the role of *initiator* in the Contract Net Protocol if and only if the other object plays the role of *participant* [3]. The symmetry of roles is closer to the traditional vision of roles as ends of a relation.

4. AFFORDANCES IN POWERJAVA

Baldoni *et al.* [3] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct defining the role with its name, the requirements and the signatures of the operations which represent the affordances of the interaction with an object by playing the role.
2. The implementation of a role, inside an object and according to its definition.
3. A construct for playing a role and invoking the operations of the role.

We illustrate powerJava by means of an example. Let us suppose to have a printer which supplies two different ways of accessing to it: one as a normal user, and the other as a superuser. Normal users can print their jobs and the number of printable pages is limited to a given maximum. Superusers can print any number of pages and can query for the total number of prints done so far. In order to be a user one must have an account, which is printed on the pages. The role specification for the user and superuser is the following:

```
role User playedby AccountedPerson {
    int print(Job job);
    int getPrintedPages();}

role SuperUser playedby AccountedPerson {
    int print(Job job);
    int getTotalPages();}
```

Requirements must be implemented by the objects which act as players.

```
class Person implements AccountedPerson {
    Login login;  // ...
    Login getLogin() {return login;}
}
```

```
interface AccountedPerson {
  Login getLogin();}
```

Instead, affordances are implemented in the class in which the role itself is defined. To implement roles inside it we revise the notion of *Java inner class*, by introducing the new keyword `definerole` instead of `class` followed the name of the role definition that the class is implementing (see the class `Printer` in Figure 2). Role specifications cannot be implemented in different ways in the same class and we do not consider the possibility of extending role implementations (which is, instead, possible with inner classes), see [4] for a deeper discussion.

As a Java inner class, a role implementation has access to the private fields and methods of the outer class (in the above example the private method `print` of `Printer` used both in role `User` and in role `SuperUser`) and of the other roles defined in the outer class. This possibility does not disrupt the encapsulation principle since all roles of a class are defined by the same programmer who defines the class itself. In other words, an object that has assumed a given role, by means of the role's methods, has access and can change the state of the object the role belongs to and of the sibling roles. In this way, we realize the affordances envisaged by our analysis of the notion of role.

The class implementing the role is instantiated by passing to the constructor an instance of an object satisfying the requirements. The behavior of a role instance depends on the player instance of the role, so in the method implementation the player instance can be retrieved via a new reserved keyword: `that`, which is used only in the role implementation. In the example of Figure 2 `that.getLogin()` is parameter of the method `print`.

All the constructors of all roles have an implicit first parameter which must be passed as value the player of the role. The reason is that to construct a role we need both the object the role belongs to (the object the construct `new` is invoked on) and the player of the role (the first implicit parameter). This parameter has as its type the requirements of the role an dit is assigned to the keyword `that`. A role instance is created by means of the construct `new` and by specifying the name of the "inner class" implementing the role which we want to instantiate. This is like it is done in Java for inner class instance creation. Differently than other objects, role instances do not exist by themselves and are always associated to their players and to the object the role belongs to.

The following instructions create a printer object `laser1` and two person objects, `chris` and `sergio`. `chris` is a normal user while `sergio` is a superuser. Indeed, instructions four and five define the roles of these two objects w.r.t. the created printer.

```
Printer hp8100 = new Printer();
//players are created Person
chris = new Person();
Person sergio = new Person();
//roles are created
hp8100.new User(chris);
hp8100.new SuperUser(sergio);
```

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, as specified by the role definition. Moreover, a role represents a specific state which is different from the player's one, which can evolve with time by invoking methods on the roles. The relation between the object and the role must be transparent to the programmer: it is the object which has to maintain a reference to its roles.

Methods can be invoked from the players, given that the player is seen in its role. To do this, we introduce the new construct:

```
receiver <-(role) sender
```

This operation allows the `sender` (the player of the role) to use the affordances given by `role` when it interacts with the `receiver` the role belongs to. It is similar to *role cast* as introduced in [1, 4, 5] but it stresses more strongly the interaction aspect of the two involved objects: the sender uses the role defined by the receiver for interacting with it. Let us see how to use this construct in our running example.

In the example the two users invoke method `print` on `hp8100`. They can do this because they have been empowered of printing by their roles. The act of printing is carried on by the private method `print`. Nevertheless, the two roles of `User` and `SuperUser` offer two different way to interact with it: `User` counts the printed pages and allows a user to print a job if the number of pages printed so far is less than a given maximum; `SuperUser` does not have such a limitation. Moreover, `SuperUser` is empowered also for viewing the total number of printed pages. Notice that the page counter is maintained in the role state and persists through different calls to methods performed by a same sender/player towards the same receiver as long as it plays the role.

```
(hp8100 <-(User) chris).print(job1);
(hp8100 <-(SuperUser) sergio).print(job2);
(hp8100 <-(User) chris).print(job3);
System.out.println("Chris printed "+
  (hp8100 <-(User) chris).getPrintedPages());
System.out.println("The printer printed" +
  (hp8100 <-(SuperUser) sergio).getTotalPages());
```

By maintaining a state, a role can be seen as realizing a *session-aware interaction* between objects, in a way that is analogous to what done by cookies on the WWW or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages, as in the above example. Note that, when we talk about playing a role we always mean playing a role instance (or *qua individual* [17] or *role enacting agent* [8]) which maintains the properties of the role.

Since an object can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when it is invoked. It is sufficient to specify which is the role of a given object, we are referring to. In the example `chris` can become also `SuperUser` of `hp8100`, besides being a normal `user`

```
hp8100.new SuperUser(chris);
(hp8100 <-(SuperUser) chris).print(job4);
(hp8100 <-(User) chris).print(job5);
```

Notice that in this case two different sessions will be kept: one for `chris` as normal `User` and the other for `chris` as `SuperUser`. Only when it prints its jobs as a normal `User` the page counter is incremented.

```
class Printer {
  private int totalPrintedPages = 0;
  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages(); ... // performs printing
  }
  definerole User {
    int counter = 0;
    public int print(Job job) {
      if (counter > MAX_PAGES_USER) throws new IllegalPrintException();
      counter += job.getNumberPages();
      Printer.this.print(job, that.getLogin());
      return counter;}
    public int getPrintedPages(){ return counter; }
  }
  definerole SuperUser {
    public int print(Job job) {
      Printer.this.print(job, that.getLogin());
      return totalPrintedPages;}
    public int getTotalpages() { return totalPrintedPages; }
  }
}
```

Figure 2: The `Printer` class and its affordances

5. RELATED WORK

There is a huge amount of literature concerning roles in programming languages, knowledge representation, multiagent systems and databases. Thus we can compare our approach only with a limited number of other approaches.

First of all, our approach is consistent with the definition of roles in ontologies given by Masolo *et al.* [17], as we discuss in [4].

The term of role is already used also in Object Oriented modelling languages like UML and it is related to the notion of collaboration: “while a classifier is a complete description of instances, a classifier role is a description of the features required in a particular collaboration, i.e. a classifier role is a projection of, or a view of, a classifier.” This notion has several problems, thus Steimann [19] proposes a revision of this concept merging it with the notion of interface. However, by role we mean something different from what is called role in UML. UML is inspired by the relation view of roles: roles come always within a relation. In this view, which is also shared by, e.g., [15, 16], roles come in pairs: buyer-seller, client-server, employer-employee, *etc.*. In contrast, we show, first, that the notion of role is more basic and involves the interaction of one object with another one using one single role, rather than an association. Second, we highlight that roles have a state and add properties to their players besides requiring the conformance to an interface which shadows the properties of the player.

A leading approach to roles in programming languages is the one of Kristensen and Osterbye [14]. A role of an object is “a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects”. Even if at first sight this definition seems related to ours, it is the opposite of our approach. By “a role of an object” they mean the role played by an object, we mean, instead, the role offered by an object to interact with it. They say a role is an integral part of the object and at the same time other objects need to see the object in a certain restricted way by means of roles. A person can have the role of bank employee, and thus its properties are extended with the properties of employee. In our approach, instead, by a role of an object we mean the role offered by an object to interact with it by playing the role: roles allow objects which can interact in different ways with players of different roles. We focus on the fact that to interact with a bank an object must play a role defined by the bank, e.g., employee, and to play a role some requirements must be satisfied.

Roles based on inner classes have been proposed also by [12, 20]. However, their aim is to model the interaction among different objects in a context, where the objects interact only via the roles they play. This was the original view of our approach [1], too. But in this paper and in [3] we extend our approach to the case of roles used to interact with a single object to express the fact that the interaction possibilities change according to the properties of the interactants.

Aspect programming addresses some of the concerns we listed in the Introduction. In particular, aspect weaving allows to change the meaning of methods. Sometimes aspects are packed into classes to form roles which have their own state [12]. However, the aim of our proposal is different from modelling crosscutting concerns. Our aim is to model the interaction possibilities of an object by offering different sets of methods (with a corresponding state) to different objects by means of roles. Roles are explicitly expressed in the method call and specify the affordances of an object: it is not only the meaning of methods which changes but the possible methods that can be invoked. If one would like to integrate the aspect paradigm within our view of object oriented programming, the natural place would be use aspects to model the environment where the interaction between objects happens. In this way, not only the interaction possibilities offered by an object would depend on the caller of a method, but they would also depend on the environment where the interaction happens. Consider the `within` construct in Object Teams/Java [12] which specifies aspects as the context in which a block of statements has to be executed. Since, when defining the interaction possibilities of an object it is not possible to foresee all possible contexts, the effect of the environment can be better modelled as a crosscutting concern. Thus aspect programming is a complementary approach with respect to our one.

Some patterns partially address the same problems of this paper. For example, the strategy design pattern allows to dynamically change the implementation of a method. However, it is complex to implement and it does not address the problem of having different methods offered to different types of callers and of maintaining the state of the interaction between caller and callee.

Baumer *et al.* [6] propose the role object pattern to solve the problem of providing context specific views of the key abstractions of a system. They argue that different context-specific views cannot be integrated in the same class, otherwise the class would have a bloated interface, and unanticipated changes would result in recompilations. Moreover, it is not possible either to consider two views on an object as an object belonging to two different classes, or else the object would not have a single identity. They propose to model context-specific views as role objects which are dynamically attached to a core object, thus forming what they call a subject. This adjunct instance should share the same interface as the core object. Our proposal is distinguished by the fact that roles are always roles of an institution. As a consequence they do not consider the additional methods of the roles as powers which are implemented using also the requirements of the role. Finally, in their model, since the role and its player share the same interface, it is not possible to express roles as partial views on the player object.

6. CONCLUSION

In this paper we introduce the notion of affordance developed in cognitive science to extend the notion of object in the object orientation paradigm. In our model objects have attributes and operations which depend on the interaction with other objects, according to their properties. Sets of affordances form role types whose instances are associated with players which satisfy the requirements associated with roles. Since role instances have attributes they provide the state of the interaction with an object.

In [1] we present a different albeit related notion of role, with a different aim: representing the organizational structure of institutions which is composed of roles. The organization represents the context where objects interact only via the roles they play by means of the powers offered by their roles (what we call here affordances). E.g., a class representing a university offers the roles of student and professor. The role student offers the power of giving exams to players enrolled in the university. In [5] we explain how roles can be used for coordination purposes. In [4] we investigate the ontological foundations of roles. In [2] we describe the preprocessor translating powerJava into Java. In this paper, instead, we use roles to articulate the possibility of interaction provided by an object.

Future work concerns modelling the symmetry of roles. In particular, the last diagram of Figure 1 deserves more attention. For example, the requirements to play a role must include the fact that the player must offer the symmetric role (e.g., initiator and participant in a negotiation). Moreover, in that diagram the two roles are independent, while they should be related. Finally, the fact that the two roles are part of a same process (e.g., a negotiation) should be represented, in the same way we represent that student and professor are part of the same institution.

7. REFERENCES

[1] M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *LNCS 3862: Procs. of PROMAS'05 workshop at AAMAS'05*, pages 57–75, Berlin, 2005. Springer.

[2] M. Baldoni, G. Boella, and L. van der Torre. Social roles, from agents back to objects. In *Procs. of WOA'05 Workshop*, Bologna, 2005. Pitagora.

[3] M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Interaction among objects. In *Procs. of PROMAS'06 workshop at AAMAS'06*, 2006.

[4] M. Baldoni, G. Boella, and L. van der Torre. Powerjava: ontologically founded roles in object oriented programming language. In *Procs. of OOOPS Track of ACM SAC'06*, pages 1414–1418. ACM, 2006.

[5] M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science*, 150(1):9–29, 2006.

[6] D. Baumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. In *Procs. of PLOP'02*, 2002.

[7] J. Cabot and R. Raventos. Conceptual modelling patterns for roles. In *LNCS 3870: Journal on Data Semantics V*, pages 158–184, Berlin, 2006. Springer.

[8] M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Procs. of AAMAS'03*, pages 489–496, New York (NJ), 2003. ACM Press.

[9] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{II}$. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.

[10] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In *LNCS 2935: Procs. of AOSE'03*, pages 214–230, Berlin, 2003. Springer.

[11] J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlabum Associates, New Jersey, 1979.

[12] S. Herrmann. Roles in a context. In *Procs. of AAAI Fall Symposium Roles'05*. AAAI Press, 2005.

[13] T. Juan, A. Pearce, and L. Sterling. Roadmap: extending the gaia methodology for complex open system. In *Procs. of AAMAS'04*, pages 3–10, 2002.

[14] B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, 1996.

[15] F. Loebe. Abstract vs. social roles - a refined top-level ontological analysis. In *Procs. of AAAI Fall Symposium Roles'05*, pages 93–100. AAAI Press, 2005.

[16] C. Masolo, G. Guizzardi, L. Vieu, E. Bottazzi, and R. Ferrario. Relational roles and qua-individuals. In *Procs. of AAAI Fall Symposium Roles'05*. AAAI Press, 2005.

[17] C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social roles and their descriptions. In *Procs. of KR'04*, pages 267–277. AAAI Press, 2004.

[18] M. Papazoglou and B. Kramer. A database model for object dynamics. *The VLDB Journal*, 6(2):73–96, 1997.

[19] F. Steimann. A radical revision of UML's role concept. In *Procs. of UML2000*, pages 194–209, 2000.

[20] T. Tamai. Evolvable programming based on collaboration-field and role model. In *Procs. of IWPSE'02*, pages 1–5. ACM, 2002.

Experiences of using the Dagstuhl Middle Metamodel for defining software metrics

Jacqueline A. McQuillan
Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare, Ireland
jmcq@cs.nuim.ie

James F. Power
Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare, Ireland
jpower@cs.nuim.ie

ABSTRACT

In this paper we report on our experiences of using the Dagstuhl Middle Metamodel as a basis for defining a set of software metrics. This approach involves expressing the metrics as Object Constraint Language queries over the metamodel. We provide details of a system for specifying Java-based software metrics through a tool that instantiates the metamodel from Java class files and a tool that automatically generates a program to calculate the expressed metrics. We present details of an exploratory data analysis of some cohesion metrics to illustrate the use of our approach.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.7 [**Distribution, Maintenance, and Enhancement**]: [Restructuring, reverse engineering, and re-engineering]

General Terms

Design, Measurement, Standardization

1. INTRODUCTION

Software plays a pivotal role in many important aspects of modern daily life. In many cases, if software fails it can have catastrophic consequences such as economic damage or loss of human life. Therefore, it is important to be able to assess the quality of software. Software metrics have been proposed as a means of determining software quality. For example, studies have demonstrated a correlation between software metrics and quality attributes such as fault-proneness [2] and maintenance effort [12].

Many software metrics have been proposed in the literature [5, 13, 9]. In order for these metrics to be widely accepted, empirical studies of the use of these metrics as quality indicators are required. However, there is no standard terminology or formalism for defining software metrics and consequently many of the metrics proposed are incomplete, ambiguous and open to a variety of different interpretations [4]. For example, Churcher and Shepperd have identified ambiguities in the suite of metrics proposed by Chidamber and Kemerer [5, 6]. This makes it difficult for researchers to replicate experiments and compare existing experimental results and it hampers the empirical validation of these metrics.

Several authors have proposed various approaches for specifying software metrics. Briand et al. propose two extensive frameworks for software measurement, one for measuring coupling and the other for measuring cohesion in object oriented systems [3, 4]. Other approaches include the proposal of formal models on which to base metric definitions and the proposal of existing languages such as XQuery and SQL as metric definition languages [16, 8, 18]. Baroni et al. propose the use of the Object Constraint Language (OCL) and the Unified Modelling Language (UML) metamodel as a mechanism for defining UML-based metrics [1].

In this paper we present details of an approach for specifying Java-based software metrics. This approach is based on one previously proposed by Baroni et al. and involves expressing the metrics as OCL queries over a Java metamodel. We have chosen the *Dagstuhl Middle Metamodel* as our Java metamodel, and we describe the specification of software metrics over this model using OCL. We have implemented a system that supports this approach which provides a flexible and reusable environment for the specification and calculation of software metrics. The system is capable of automatically generating a program to calculate the specified metrics. We have performed a study of several cohesion measures to demonstrate its use.

The remainder of this paper is organised as follows. Section 2 gives details of the approach for specifying software metrics. Details of a system that implements this approach are presented in Section 3. In Section 4, we present an exploratory data analysis of some cohesion metrics. Section 5 gives conclusions and discusses future work.

2. DEFINING METRICS

In this section, we give details of an approach for specifying Java based software metrics that is based on the use of metamodels and the OCL.

As the name suggests, a *metamodel* is a model that describes other models. Typically, we think of a model of a software system as being a design model, such as UML class or sequence diagrams, or an implementation model, such as an actual program. A metamodel then would describe

```
-- Returns the RFC value for the Class c
context ckmetricset::RFC(c:DMM::Class) : Real
body: self.implementedMethods(c)->union(self.methodsDirectlyInvoked(c))->asSet()->size()

-- Returns a set containing all methods directly invoked by all the implemented methods of Class c
def: methodsDirectlyInvoked(c:DMM::Class) : Set(DMM::Method)
  = self.implementedMethods(c)->collect(m:DMM::Method | self.methodsDirectlyInvoked(m))
                               ->flatten()->asSet()

-- Returns a set containing all methods directly invoked by the Method m
 def: methodsDirectlyInvoked(m:DMM::Method) : Set(DMM::Method)
   = m.invokes->select(be:DMM::BehaviouralElement | self.isKindOfMethod(be))
             ->collect(belem:DMM::BehaviouralElement | belem.oclAsType(DMM::Method))->asSet()
```

Figure 1: RFC Metric Definition using the DMM. ***This OCL specification defines an operation to calculate the RFC metric for a class, as well as two auxiliary operations. The entities used in the definition are from the DMM.***

the allowable constructs in these models and the allowable relationships between these constructs.

OCL is a standard language that allows constraints and queries over object oriented models to be written in a clear and unambiguous manner [17]. It offers the ability to navigate over instances of object oriented models, allowing for the collection of information about the navigated model. Baroni et al. propose expressing design metrics as OCL queries over the UML 1.3 metamodel [1]. Their approach involves modifying the metamodel by creating the metrics as additional operations in the metamodel and expressing them as OCL conditions.

We have already extended the approach of Baroni et al. to the UML 2.0 metamodel in a manner specifically designed to be reusable for other metamodels [14]. Our extension involves decoupling the metric definitions from the metamodel by creating a metrics package at the meta level. Defining a new metrics set is then a three step process. First a class is created in the metrics package corresponding to the metric set. Then, for each metric, an operation in the class is declared, parameterised by the appropriate elements from the metamodel. Finally the metrics are defined by expressing them as OCL queries using the OCL `body` expression.

2.1 Selecting a metamodel

In order to adapt this approach to specify Java based metrics, it is necessary to have a model of Java programs i.e. a Java metamodel. There is no standardised Java metamodel, but a number of task-specific metamodels have been proposed for various purposes. In order to maximise the reusability of our metrics, we have chosen to use the *Dagstuhl Middle Metamodel* (DMM) as a basis for defining our metrics [11]. The DMM was designed as a schema for reverse engineering that would facilitate interoperability between tools as an agreed exchange format. It is a “middle” metamodel in so far as it seeks to be more abstract than a syntax graph, but less abstract than a high-level architectural description.

The DMM itself is language independent, but contains many features commonly found in languages such as C, C++, Java and Fortran. We do not have space here to reproduce the elements of the model, necessary for a full understanding of our metrics, but details can be found in [11]. We have used the Chidamber and Kemerer (CK) metrics suite [5] to illustrate the approach outlined in this paper. We have successfully expressed the CK metrics as OCL queries over classes from the `ModelObject` hierarchy of version 0.007 of the DMM. This required approximately 29 OCL queries in total.

As an example of a metric definition, Figure 1 presents the definition of the response set for a class (RFC) metric. The response set for a class is the set of all implemented methods of this class and all methods invoked by this class. The definition is parameterised by a single `Class` from the DMM hierarchy, and the body of the definition returns the size of the response set for this class. The auxiliary operation `methods-DirectlyInvoked(DMM::Class c)` gathers all methods invoked by each of the implemented methods in the class. The operation `methodsDirectlyInvoked (DMM::Method m)` traverses the invokes association in the DMM to gather all `BehaviouralElements` invoked by the method `m` and then selects all elements from this set that are `Methods`.

3. IMPLEMENTATION

In this section we describe the implementation of our system to calculate metrics for Java programs based on the DMM. This was a three step process:

Step 1: Create a representation of the classes and associations of the DMM in Java

Step 2: Develop a tool to convert Java programs to instances of the DMM

Step 3: Develop a tool that can apply metrics defined in OCL to the instance of the DMM produced in step 2.

Step 1 is easily achieved by depicting the DMM as a UML class diagram, and then using the Octopus [10] tool to generate the corresponding Java classes. We implemented the 19 classes from the DMM `ModelObject` hierarchy directly, and chose to implement the relationships using attributes of these classes, rather than association classes. This implementation decision was made as the explicit relationship classes were not required by our tool. For similar reasons, we did not implement the classes in the `SourceObject` hierarchy that represent details about the code as it appears in the original program. This does not preclude these classes being added later.

The combination of the tools used in our approach is shown in Figure 2. The figure is divided into two layers: the upper layer represents the metric definition process, which

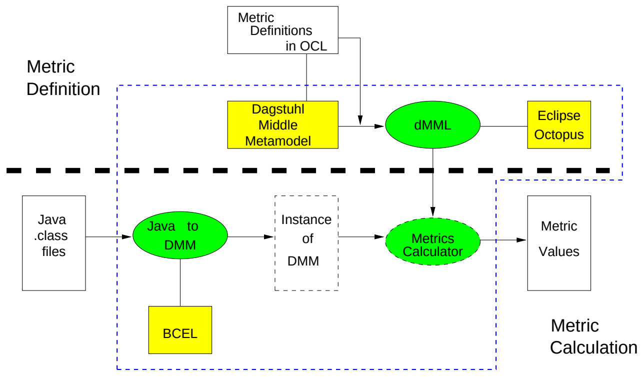


Figure 2: The use of dMML to define and calculate metrics for Java programs. dMML *is part of a tool-chain that calculates metric values from Java .class files.*

is done once for each metric set. The lower layer represents the metric calculation process, where the metrics are applied to a set of Java programs. The main tools we developed are shown as green ovals: dMML for metric definitions, and *Java to DMM* for converting class files to instances of the DMM. The *Metrics Calculator*, also shown as a green oval, is a Java program automatically generated by dMML for each metric set. The third-party software used in our metric definition system is shown by yellow boxes in Figure 2. BCEL is used by the *Java to DMM* tool, and the Octopus plug-in for Eclipse is used in defining the metrics, as described in the following two sub-sections. The definition of the DMM is represented as a UML class diagram, and the corresponding Java representation is forward-engineered using Octopus.

The blue dashed line in Figure 2 delimits the system, and shows that its inputs are a set of metric definitions in OCL and a set of Java programs. The output of the system is the set of metric values calculated by applying the metrics to the Java programs.

3.1 Converting Java programs to an instance of the DMM

In order to complete Step 2, it is necessary to read in Java programs and to instantiate the DMM classes produced in Step 1. We chose to process a compiled `.class` file directly as the contents of the `.class` file most closely resembled the information needed to instantiate the DMM implementation. In particular, access relationships between classes arising from the use of fields and variables in a method are easy to identify at the bytecode level, since they are translated into a single bytecode instruction.

Our implementation uses the Apache Bytecode Engineering Library (BCEL) to read in and traverse the contents of the `.class` file. The BCEL API provides classes representing the contents of the `.class` file, and methods to access classes, fields, methods and bytecode instructions. Using the BCEL it was relatively easy to traverse these structures and instantiate the DMM, and required less than 600 (non-blank, non-comment) lines of Java code. It should be noted that using BCEL would not be suitable for a more detailed representation than the DMM `ModelObject` hierarchy. Source level details such as Java statements (e.g. while and for loops) are not represented in the bytecode, and tables giving local variable names and mappings to lines of Java code are optional at the `.class` file level.

3.2 Implementing the metric definitions

To complete step 3, we extended a prototype tool dMML (for *Defining Metrics at the Meta Level*) that was first applied to the UML 2.0 metamodel [14]. Our tool is implemented as a plug-in for the integrated development environment Eclipse.

In step 3, a set of metrics are created and defined for the language under consideration which in this case is Java. To achieve this, the language metamodel, DMM is provided along with the metric definitions expressed as OCL queries over this metamodel. dMML uses the Octopus plug-in to perform syntactic and semantic checks on these OCL expressions.

The dMML tool uses the Octopus plug-in to translate the OCL metric definitions into Java code. A Java program, *Metrics Calculator*, is automatically generated by dMML that will calculate the defined metrics for any instance of the Java metamodel (i.e. Java programs). To perform the metric calculations dMML uses the *Java to DMM* tool developed in step 2 to convert the Java programs to an instance of the DMM. The Java code corresponding to the metric definitions is executed and the results from the metric calculations are exported in text format.

The parameterisation of the dMML tool by both the language metamodel and the definition of the metrics set is an important feature of our approach. To extend our system to work with other language metamodels only step 2 of this process needs to be changed. That is, a new tool would need to be developed to create instances of the language metamodel.

4. EXPLORATORY DATA ANALYSIS

In this section we present an exploratory data analysis of cohesion measures in order to demonstrate the feasibility of our approach to defining and implementing software metrics.

Using the procedure described in previous sections we have implemented several cohesion measures from [3] and applied them to programs from the DaCapo benchmark suite, version *beta051009* [7]. This benchmark suite is designed for memory management research, and consists of 10 open-source real-world programs.

In order to provide a meaningful comparison, we have chosen the four cohesion metrics that do not involve indirect comparisons, namely $LCOM1$, $LCOM2$, $LCOM5$ and ICH [3]. We have included constructors, finalisers and accessor methods as ordinary methods, but excluded attributes and methods that are inherited but not defined in a class. Since metric $LCOM5$ involves division, we have excluded those classes that cause a divide-by-zero error, namely classes that contain no attributes, or classes that contain exactly one method definition. A total of 4836 classes in the DaCapo benchmark suite meet these conditions.

Table 1 gives a summary of the values of the four metrics over these 4836 classes. The values of $LCOM1$ and $LCOM2$ are all positive integers, whereas $LCOM5$ is normalised to a real number between 0.0 and 2.0. The measure ICH has been negated to facilitate comparison since it measures the degree of cohesion, rather than the lack of cohesion measured

	LCOM1	LCOM2	LCOM5	ICH
Min.	0.0	0.0	0.0000	-3887.0
1st Qu.	2.0	0.0	0.5000	-10.0
Median	11.0	6.0	0.8000	-2.0
Mean	185.5	135.5	0.7063	-20.5
3rd Qu.	52.0	34.0	0.9444	0.0
Max.	110902.0	94039.0	2.0000	0.0

Table 1: Summary of the values for the metrics applied to 4836 of the classes from the DaCapo suite. ***This table shows the minimum, maximum and mean data values, as well as those at the first, second (median) and third quartiles.***

	LCOM1	LCOM2	LCOM5	ICH
LCOM1	1.0	0.8597	0.5305	-0.7439
LCOM2	0.8597	1.0	0.6453	-0.6463
LCOM5	0.5305	0.6453	1.0	-0.3739
ICH	-0.7439	-0.6463	-0.3739	1.0

Table 2: Spearman's ρ statistic for each pairing of the four metrics. ***This statistic compares data sets on a rank basis, with values near 1.0 or -1.0 indicating a strong positive or negative correlation.***

by the $LCOM$ metrics. The values of ICH are thus all negative integers. That the values shown in Table 1 all fall within these theoretical bounds provides a coarse-grained validation of our implementation.

Figure 3 shows a histogram of the distribution of values for each of the four metrics. In this figure, we have removed the last 5% of data values, including some extreme outliers, in order to better visualise the distribution. The values for $LCOM1$, $LCOM2$ and ICH are all heavily skewed towards zero, while the normalised values for $LCOM5$ reflect a somewhat more even distribution. The distribution of the metric values is consistent with previous work, which showed a similar preponderance of low values for the SPEC JVM98 and JavaGrande benchmark suite [15], and reflects known limitations of these cohesion metrics [2]. The percentage of classes giving a value of zero was 16% for $LCOM1$, 31% for $LCOM2$, 11% for $LCOM5$, and 39% for ICH.

In order to check for a relationship between the metric values, the pairwise scatter plots were examined and Spearman's ρ statistic was used to estimate the level of association. We have omitted the scatter plots to save space, but the results for Spearman's ρ are shown in Table 2. This statistic compares data sets on a rank basis: thus, values close to 1.0 (or -1.0) indicate that the metrics are ranking the classes in the same (or opposite) order. The results in Table 2 confirm the most obvious relationships, i.e. that there is a strong positive correlation between the measures $LCOM1$ and $LCOM2$, which have similar definitions, and that each of the LCOM measures have a negative correlation with ICH, although this is weak for $LCOM2$ and $LCOM5$. The strong (negative) association between the values for $LCOM1$ and ICH is interesting, since these metrics are based on quite different definitions. Further investigation would be required to see if this is a general property of these metrics.

The results presented in this section are exploratory in nature, and only provide a preliminary coarse-grained de-

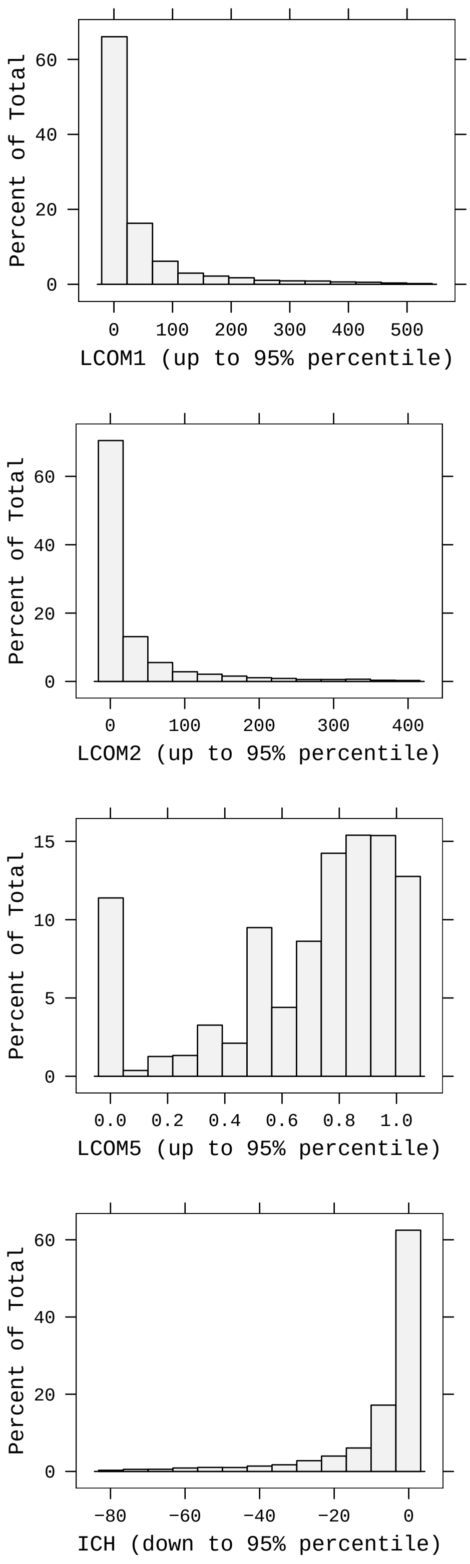


Figure 3: Histograms showing the distribution of values for the four cohesion metrics. ***For clarity, only the first*** 95% ***of the values are shown in order to remove extreme outliers.***

scription of the metric values. Nonetheless, we believe that providing such data is important in order to demonstrate the robustness of the metric calculation tool, and as a "smoke test" to ensure that the values are within reasonable boundaries. The design of the dMML tool facilitates the definition of multiple metrics suites, and we hope to exploit this in order to assemble a substantial database of descriptive statistics of object-oriented metrics for benchmark programs.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have harnessed the OCL as a language to specify metric definitions over the *Dagstuhl Middle Metamodel* . We have implemented a system that uses the Octopus tool to translate OCL metric definitions into Java code and then calculates the metrics for a Java program. To demonstrate the feasibility of our approach, we have specified four of the alternative definitions of cohesion in OCL and used our system to calculate the metrics for a suite of 10 real-world programs.

In previous work we have applied this approach to defining outline metrics over class diagrams from the standardised UML metamodel [14]. However, metrics defined at the class diagram level cannot evaluate features internal to methods, such as number of method calls etc. A key tenet of our approach is that the range and variance among metric definitions requires a flexible and reusable definition environment.

While developing and implementing the metric definitions and the associated dMML tool, a number of issues arose which we hope to deal with in future work.

- First, despite the formal definition of metrics in [3], there are still some ambiguities for trivial and extreme cases of the metrics, such as when there are no attributes or no methods in a class.
- Second, the correctness of the metric definitions hinges on assumptions made while constructing the metamodel. For example, our tool to translate a class file to an instance of the DMM does not include inherited methods in a class definition. Therefore, it is important that such assumptions would be a known, expressed feature of any metric definition framework.
- Third, the correctness of the program to translate classes to instances of the DMM has not been verified. Errors or omissions at this stage would have a fundamental impact on the correctness of the calculated metrics.

We intend to continue our work by developing a full set of coupling and cohesion metrics, applied to both a Java and the UML metamodel, and to investigate the full potential of modularity and re-usability associated with defining metrics at the meta level.

6. REFERENCES

[1] A. Baroni, S. Braz, and F. Brito e Abreu. Using OCL to formalize object-oriented design metrics definitions. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.

[2] V. Basili, L. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[3] L. C. Briand, J. W. Daly, and J. K. Wuest. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[4] L. C. Briand, J. W. Daly, and J. K. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] N. Churcher and M. Shepperd. Comments on 'A metrics suite for object-oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, 1995.

[7] A. Diwan, S. Guyer, C. Hoffmann, A. L. Hosking, K. S. McKinley, J. E. B. Moss, D. Stefanovic, and C. C. Weems. The DaCapo project. http://www-ali.cs.umass.edu/DaCapo/, Last accessed July 17, 2006.

[8] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy. A novel approach to formalize object-oriented design metrics. In *Evaluation and Assessment in Software Engineering*, Keele, UK, April 2005.

[9] N. Fenton and S. Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thompson Computer Press, 1996.

[10] Klasse Objecten. Octopus: OCL tool for precise UML specifications. Available from http://www.klasse.nl/octopus/, Last accessed July 17, 2006.

[11] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl Middle Metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, May 10 2004.

[12] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[13] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics.* Prentice Hall Object-Oriented Series, 1994.

[14] J. A. McQuillan and J. F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming*, Nantes, France, July 2006.

[15] Á. Mitchell and J. F. Power. Run-time cohesion metrics for the analysis of Java programs - preliminary results from the SPEC and Grande suites. Technical Report NUIM-CS-TR2003-08, Dept. of Computer Science, NUI Maynooth, April 2003.

[16] R. Reißing. Towards a model for object-oriented design measurement. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Budapest, Hungary, June 2001.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison-Wesley, 2003.

[18] F. Wilkie and T. Harmer. Tool support for measuring complexity in heterogeneous object-oriented software. In *IEEE International Conference on Software Maintenance*, Montréal, Canada, October 2002.

Reducing Java Internet project risks: a case study of public measurement of client component functionality in the user community

Tomas Hruz
Institute for Theoretical Computer Science
ETH Zurich
8092 Zurich, Switzerland
tomas.hruz@inf.ethz.ch

Matthias Hirsch-Hoffmann
Institute of Plant Sciences
ETH Zurich
8092 Zurich, Switzerland

Wilhelm Gruissem
Institute of Plant Sciences
ETH Zurich
8092 Zurich, Switzerland

Philip Zimmermann
Institute of Plant Sciences
ETH Zurich
8092 Zurich, Switzerland

ABSTRACT

A major risk for Internet software projects which have server and client components are decisions related to availability and features on client computers in the user community. Specifically, bioinformatics software developers intending to use Java face critical decisions about which Java version to implement, but few statistics are available about Java presence on user machines. To obtain this information, we implemented a measurement system to detect the presence, functionality and version of Java Virtual Machines on client computers of a large base of users from the biology community. We show that our system effectively collects the necessary information and provides decision-relevant statistics. Measurements performed on 1753 client computers showed that Java presence is high and dominated by the most recent Java versions. The proposed empirical approach can be used to reduce decision risks in any type of Internet software project with low level of control on client equipment and high demands on client interaction and performance. More details together with source code and measurement results can be obtained from the J-vestigator survey page (https://www.genevestigator.ethz.ch/index.php?page=jvestigator)

1. INTRODUCTION

Most Internet software projects which have a client software component face critical decisions in the early project phase when assumptions about client computers in the user community have to be taken. A distinctive feature of Internet software projects compared to other industrial software productions is that the platforms and features installed on the user computers are not centrally controllable and software project decision makers can influence the user community only in a very limited way.

The present case study illustrates a method about how to reduce the above-mentioned risk using a measurement system which provides systematic data about components installed on user computers in the community where the new software is supposed to be used. Once the structure of components installed on these computers is known, a fact-based decision concerning client software technologies can be made. Moreover, our method classifies user computers to well defined groups of potential problems that users will encounter when using the client software product. The product penetration and functionality can therefore be increased with a targeted information campaign for such user groups, providing detailed explanations and procedures how to install or upgrade necessary components on their computers.

More specifically, the problem of deciding about a client software technology in a distributed client/server system has three important parameters leading to a three dimensional decision space. The first parameter represents the complexity of interaction on the client side of the application. If a very complex interaction is projected, a more powerful programming platform like Java compared to Web browser is necessary. The second parameter is the server load. A powerful programming language on the client side is needed if users repeat with high frequency cpu intensive computing. In such cases, a lack of possibility to compute on the client side leads to intractable load on the servers. The third important parameter is the user environment controllability. This describes to what extent the application provider can control which platform features, and which versions are installed on user computers. In large companies with homogeneous configuration control, application providers can prescribe the environment and therefore possess a very high client computer environment controllability. In Internet, however, the controllability is almost null. Normally, it is advised to use a platform like Java only in situations where complex interactivity and/or high server load are combined

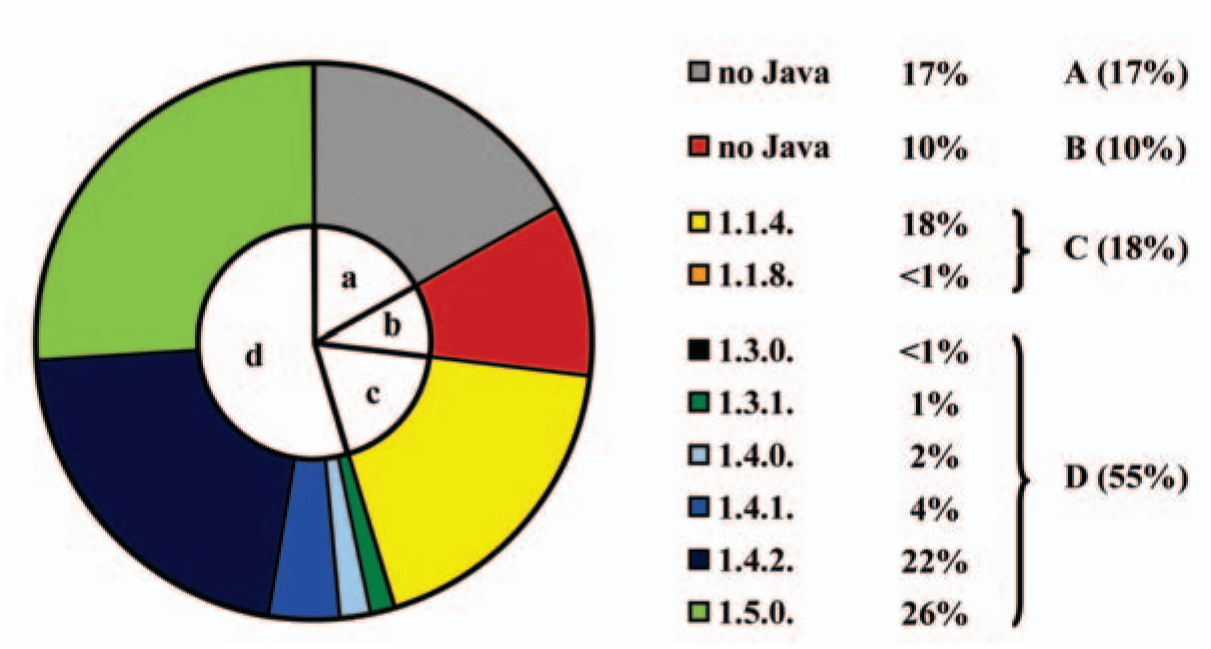


Figure 1: Results from the J-vestigator survey as of December 2005.

with high user computer environment controllability.

The presented method is directed to exactly the opposite situation, where low user environment controllability is combined with high server load and/or complex interactivity of the application. The correct measurement of software parameters on client computers can improve software development decisions, resulting in optimal choices for platforms and versions and to powerful applications increasing user satisfaction in the Internet community.

The method described in our article can be seen as an essential step on the path towards advanced and functionally rich Internet software applications. In our specific case, the correct identification of the future technology required the presence of a more simple web interface application. Generally, if we consider the development of an Internet application in a wider frame it has analogies with the "bootstrapping" process as known in the Unix development paradigm. One way how to incorporate the method proposed in this article in a technology bootstrapping is as follows: 1) An application version based on lower level technology (e.g. html, web interface) is used. This application version defines the user community and allows to measure the more advanced technology (e.g. Java). 2) The higher level technology features are identified with a measurement step. 3) The application version is replaced/enhanced with a version containing the new technology identified in step 2 and the steps starting at 1. are repeated.

2. JAVA TECHNOLOGY DECISIONS IN BIOINFORMATICS

The challenge of bioinformatics developers is to create widely distributable, highly performant and rapidly evolving tools that meet the needs of research biologists. Java [3] is emerging as a key player in bioinformatics due to its platform independence and its object-oriented programming nature, allowing to model highly complex biological information. Although traditionally the language of choice for many bioinformaticians has been Perl, more and more applications are being developed using Java technology, frequently connected to a relational database (e.g. [2, 1]).

The environment on user computers is highly heterogeneous. First, large variations occur with respect to hardware, software platform, operating system, and installed applications. Second, Java usage is strongly related to the installed browser technology, resulting in a panoply of environments in which the application must run (see the measurement results). Despite its many advantages, web-based Java applications request compatible versions of Java on client computers.

Furthermore, as Internet-based bioinformatics applications grow in complexity and in the number of users, resources required from servers may outgrow the capacities of many server infrastructures. A reasonable solution is to translocate parts of the data processing to the client computers. For this, Java technology can provide a powerful solution if it is correctly running on the client computers.

With respect to the high investment in resources for software development and the low level of control over client computers from the Internet, initial technological decisions represent a major risk factor for software developers. Knowledge about Java penetration and functionality in the biology community represents a considerable benefit in terms of investment, strategic planning, and software distribution for bioinformatics projects using Java.

While designing a new version of Genevestigator ([4]) with the goal of optimal performance and maximum compatibility, we faced the decision about whether to use JAVA, and if yes, which version would be most suitable. In this respect, two options are available: a) to program for Java 1.1.X, with which programs will run for a larger group of users.

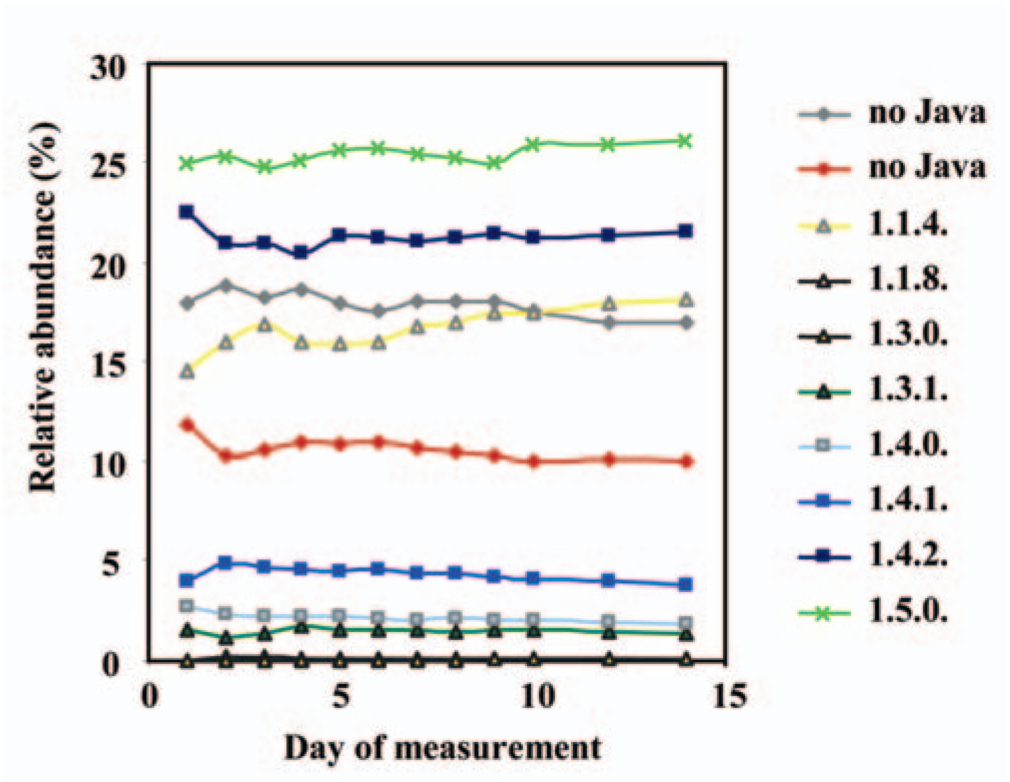


Figure 2: Evolution of the relative abundances of Java versions during the period of measurement (cumulative).

However, more difficulties occur at all levels of the product development cycle (programming, debugging, deployment etc.) because Java 1.1.X has less powerful libraries and a less efficient Virtual Machine, or b) to program for Java 1.3 or higher (referred to here as Java 2), with which the graphics and GUI elements are more powerful, the Virtual Machine is very fast, and the development tools are better integrated.

Currently, there are no public reliable statistics about Java presence, version and functionality on user computers in the biology community. To address this problem, we designed a system to measure such statistics built on Genevestigator, which has a large and world-wide user community from bioinformatics and biological research.

3. MEASUREMENT METHOD AND RESULTS

The Java measurement system consists of several layers. First, a MySQL database instance is required to store the data. Second, PHP scripts generate the appropriate Javascript code, the HTML code and a simple Java Applet to read and save the Java version of client computers. The Applet code is kept as simple as possible to be executable on all possible Java versions, and it only reports information if it runs correctly on the client machine. Therefore, to obtain information also for those cases in which Java is non-functional or absent, and to ensure that measurements can be obtained from most user computers, we developed and tested a combination of PHP, HTML, Javascript, and Java. More details about the code fragments and implementation can be obtained from the J-vestigator survey page (https://www.genevestigator.ethz.ch/index.php?page=jvestigator).

The results from the analysis of 1753 user computers reveal multiple interesting aspects. First, the complexity of the application landscape on user computers was surprising, with 371 different browser and operating system variations in the test data. This means that on average only about 5 computers share the same configuration in the pool of 1753 computers. Second, with respect to Java, the results can be grouped into four categories (see Fig. 1): A and B) no Java is available (17% and 10%, respectively), C) a functional, lower version of Java (1.1.X) is available (18%), and D) a functional, higher version of Java (Java 2) is available (55%)

Altogether, Java was present and functional on 73% of client computers (see Fig. 1, groups C and D). Considering different Java versions, Java 2 had a larger penetration than Java 1.1.X versions (55% versus 18%). In most of the latter cases, this reflects a per default implementation of Java 1.1.X in the Microsoft IE browser. As compared to Java 2, programming for Java 1.1.X would therefore add only 18% to the set of user computers for which the users must not take any action to run a Java application. Alternatively, the availability of free Java software and good manuals can help to achieve a smooth transition of such users to Java 2.

Group A (17%) represents computers in which the <applet>tag implementation does not allow Javascript to be executed inside the tag if Java is not present. In this case, an error is subsequently generated in Javascript code; however, we catch this effect and store the correct information in the database. This group contains mainly IE 6 browsers on Windows and IE 5 on Mac, both without any Java installed.

Group B (10%) represents computers in which the <applet>tag does allow Javascript but Java was not available to the browser. This can occur for the following three reasons: i) there is no Java installed on the computer, ii) the Java installation is not integrated into the browser or iii) it is deactivated by the browser configuration. For reason i) there are two major cases. First, some MS Windows versions are not provided per default with Microsoft Java Run-time. Second, Firefox browsers come per default without any Java Run-time.

To assess the robustness of our statistics, the relative abundances of each category during the period of measurement is plotted against time (see Fig. 2). After some minor initial fluctuation, the proportions tend to become stable after two weeks of measurements. More recent and longer-term robust statistics can be obtained from the J-vestigator web site.

4. CONCLUSIONS

From our case study we can draw several conclusions. Specifically, for the bioinformatics user community we conclude that Java is a good option for programming of more advanced bioinformatics applications because of its wide presence, modern design and powerful network concept. Probably the most interesting fact is the big penetration of the recent versions of Java (1.4. and 1.5) in the biology community. The decision problem (between Java 1.1.X and Java 2) which led us to the presented work is not specific for bioinformatics, however the decision result itself depends very much on the Java versions structure for this particular community.

From the general software development point of view the proposed method has brought more predictability in the decision process in the early phases of our project. We know much more about our users and about the environment where the application will be running. This knowledge gives us also a possibility to address the specific user groups with tailored help procedures. Therefore we propose to include a similar measurement step in every Internet software project where nontrivial client functionality is needed.

Moreover, with today's possibility to build similar measurement systems as shared platforms, where users share their data and know-how, crystallization kernels of a larger distributed measurement system can be provided, which would collect and share various characteristics of the Internet software environment on the client computers. Such knowledge (or even control) would reduce development risk and improve the functionality of Internet applications.

To start the work on the above idea we provide software developers and users with up-to-date statistics about Java running on client machines in a public survey which will serve as a continuous Java statistics exchange platform. Developers using the proposed measurement system in other communities are welcome to share their results through J-vestigator. Measured data are free to download over the Internet at the J-vestigator Web page.

5. REFERENCES

[1] B. Dysvik and I. Jonassen. J-express: exploring gene expression data using java. *Bioinformatics*, 17(4):369–370, 2001.

[2] J. e. a. Johnson. Tableview: portable genomic data visualization. *Bioinformatics*, 19(10):1292–1293, 2003.

[3] S. Microsystems. Java.sun.com: The source for java developers. *http://java.sun.com*, December 2005.

[4] P. e. a. Zimmermann. Gene-expression analysis and network discovery using genevestigator. *Bioinformatics*, 10(9):407–409, 2005.

Teaching Inheritance Concepts with Java

Axel Schmolitzky
University of Hamburg, Germany
Vogt-Koelln-Str. 30
D-22527 Hamburg
+49.40.42883 2302
schmolitzky@acm.org

ABSTRACT

In teaching object-oriented programming, teaching inheritance is the most challenging and at the same time the most crucial aspect. The interplay of dynamic binding, late-bound self-reference, subtype polymorphism and method redefinition is so complex that it is difficult for instructors to design a gentle, step-by-step introduction. Should polymorphism be introduced first? Or is code reuse better suited as an introductory motivation? The Java Programming Language adds a further aspect to this discussion: when should named interfaces be introduced? Most textbooks follow the historical development of the mechanism and cover interfaces after the discussion of abstract classes. In this paper a different approach is described: interfaces are introduced long before and in isolation from inheritance; and the discussion of inheritance is explicitly split into its two major constituents, namely subtype polymorphism and implementation inheritance. We applied this novel approach in the two introductory courses on software development (SD1 and SD2) in the newly created *Bachelor of Science in Informatics* curriculum at the University of Hamburg, Germany. This experience report reflects on the design rationale behind this new approach.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education*

D.1.5 [**Programming Techniques**]: Object-Oriented Programming.

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Interfaces, Polymorphism, Inheritance.*

General Terms

Languages

Keywords

Pedagogy

1. INTRODUCTION

Teaching inheritance is challenging. The (commonly assumed) basic principle, hierarchies of terms building taxonomies, is easy to explain. But when it comes to the details of the underlying programming language concepts, things are getting harder. One reason for the difficulties lies in the nature of inheritance: it is a powerful mechanism that can be used for several and quite diverse purposes. It is not easy to say which of these is more important and should be taught first (following the pedagogical pattern Early Bird, see [1]).

In the research literature on object-oriented programming languages and principles, the distinction between hierarchies of types (supporting inclusion polymorphism and subtyping) and hierarchies of implementations (aka inheritance, supporting reuse of code and guaranteeing common behavior) is well-established since the early 1990s [5, 7, 12, 15]. But in most textbooks on object-oriented programming in Java, this distinction is still not well covered (see, for example, [4, 10, 21]). The few textbooks that do distinguish the two concepts (e.g. [2]) still treat interfaces in Java according to their historical and technical roots: as fully abstract classes, after the discussion of abstract classes and abstract methods. One notable exception is the textbook by Horstmann [9]; it covers interfaces before inheritance, but it is targeted at advanced courses in the second or third semester.

In [16] we pointed out that it is reasonable to cover interfaces before inheritance, but with only little constructive advice how to do it; in this paper we provide more background on this reasoning and describe the structure of a CS1 course that exemplifies it. We further show how type and implementation hierarchies can be taught in strict sequence, again backed up by a description of a course structure that realizes it.

2. BACKGROUND

The Software Engineering Group (SWT) at the University of Hamburg is currently responsible for the first two programming modules (covering imperative and object-oriented programming) in the newly created *Bachelor of Science in Informatics* curriculum. These modules are called *Softwareentwicklung* (Software Development) *I* and *II*. We refer to them as SD1 (first semester) and SD2 (second semester), respectively. SD1 primarily covers programming (and unit testing), whereas SD2 extends to object-oriented modeling and software development processes. SD1 and parts of SD2 roughly compare to CS1 and CS2 in the North American curriculum. Both modules currently use Java as the programming language, as the SWT group has several years of experience in using this language for undergraduate and graduate education.

3. BASIC TERMINOLOGY

As object-oriented programming has no formal grounding (as compared to functional programming with the lambda calculus), we first define our understanding of some important terms for the following discussion.

3.1 Language Concepts and Mechanisms

We distinguish programming language *concepts* from programming language *mechanisms*, as exemplified in [6] for inheritance concepts and mechanisms. Language concepts are language-independent notions of programming; they are valid for all programming languages or at least for a family of languages and they abstract from specific details in different languages. Language mechanisms, on the other hand, are concrete mechanics in specific languages with formal syntax and (hopefully) exact semantics. The same concept can be supported by different mechanisms in different languages. Take the *imperative loop* as a simple example of a concept; it can be described as "a sequence of statements being repeated as long as some condition is true". The `while`- and `repeat`-loops in Pascal and the `while`- and `for`-loops in Java are examples of mechanisms or constructs implementing this concept.

One mechanism in one language can support several concepts; the mechanism is then overloaded. If the concepts that a mechanism is overloaded with are too diverse, maintaining programs written in that language can become difficult; one can not tell directly from looking at a specific usage of the mechanism which concept is meant to be supported. The inheritance mechanism in Eiffel is a good example of a mechanism that is overloaded with several concepts: type hierarchies, code inheritance, type abstraction, sometimes even simple use of services (as described in the shopping list approach in [14], the first edition of Meyer's classic). While computer science often tends to seek for generalization and unification ("everything is a function" or "everything is an object") to simplify languages, software engineering typically advocates language mechanisms that clearly transport their intended meaning for pragmatic reasons: to simplify programming *and* maintenance.

3.2 *Interfaces* and `Interfaces`

In the following we distinguish the *concept* interface from the *mechanism* interface by setting the terms in different fonts:

- The concept *interface* (set in italics in the following) denotes the conceptual interface of a class, i.e. all the methods that are public and thus available for (regular) clients of the class. The *interface* of a class should describe what instances of the class can do, not how they do it.
- The mechanism `interface` (set in typewriter font in the following) is a construct of Java (and of other languages as well) that allows to describe just the signatures of the operations of a type. An `interface` can be used to explicitly describe the complete *interface* of a class or it can be used to partially describe a role that a class can play in certain contexts [17].

Note that we do not consider yet another meaning of the term interface in this discussion: that of a graphical user interface.

3.3 Types and Classes

Classically, as described by Hoare in [8], a type is a *set of values* plus *operations* on these. In object-oriented programming, the notion of a value set is abstracted to a *set of elements* (either values or instances). A class then defines a type in the sense that the instances of the class form its (open) set of elements, and the public methods of the class (its *interface*) are the operations on these elements.

A type is a more abstract notion that can be extracted from an existing class; vice versa, a formal or informal description of a type can be taken as a specification for the implementation of a class. We set type names in italics in the following and class names in typewriter font whenever we want to stress the difference between the abstract notion of a type and the concrete implementation in the form of a class.

3.4 Operations and Methods

As soon as the concept of a type is introduced explicitly, *operations* and *methods* can be distinguished systematically. Operations are part of a type and describe the abstract notion of operations on the elements of a type's domain. Ideally, operations describe *what* can be done with the objects of a type, not *how* it is done. Methods on the other hand are structural parts of classes and contain code. The public methods of a class form the operations of the type the class defines, thus private methods are not operations in this sense. In the following we use italics for naming operations and typewriter font for naming methods.

The advantages of these distinctions become evident in connection with inheritance. Consider the method `equals` of the Java class `Object`. The type *Object* defines the operation *equals*, the class `Object` also supplies an implementing method of this operation. Subclasses of `Object` that redefine `equals` do not offer an additional operation, they just offer alternative methods for still the same operation.

3.5 Consuming and Producing

Not all concepts that have to be used by the students can be fully explained at first encounter. It is often a good pedagogical strategy to present just one aspect of a new concept, let the students make their own experiences and turn back later to fill the gaps. The Pedagogical Pattern Project identified this as the Spiral pattern [1]. We use this pattern systematically by distinguishing between *consuming* and *producing* a concept. Consuming typically comes before producing. Before a writer can write a good book, she should have read a lot of books. Before a carpenter can design a good table, he should have used several tables. Typically consuming is also easier than producing, e.g. reading a book is easier than writing a book.

We observed that this principle can be applied well to programming concepts, e.g.:

- Students *consume* the concept of *packages* when importing classes or interfaces from other packages (e.g. `List` from package `java.util` for Java). They need not know much about packages in Java, just that they are bundles of library code. When they *produce* packages through dividing larger systems into several packages, they need to know all details about package visibility and the like.

- Students *consume* genericity when they first use collections from the Java Collection Framework, which may be relatively early on. Defining the element type of a collection is quite straightforward once the concept of a type is understood. *Producing* a generic class, on the other hand, requires much more knowledge about genericity; this can be taught much later.

We applied this principle in these and in several more situations in SD1. We will now turn to this course in more detail.

4. TEACHING INTERFACES BEFORE INHERITANCE

In this section we describe the structure of SD1 in as much detail as is necessary to point out our objective: to demonstrate that teaching `interfaces` before inheritance can be done systematically.

We follow an *Objects First* approach in SD1, as described for example in [2]. Using BlueJ [11], students work with objects from day one. BlueJ is a free IDE tailored for teaching object-oriented programming that we have been using in introductory programming courses for several years now. Instances of classes can be created interactively; any of their methods can be called interactively and their states can be inspected easily. The classes of a project are visible all the time in an interactive class diagram.

4.1 *Interfaces* Early

In their first programming tasks, students just create objects of provided classes. They interact with these objects, calling the public methods defined by the given classes. For our discussion, the class editor's ability to show two different views of a class definition is an important feature of BlueJ: the implementation and the *interface* of the class (generated on the fly using javadoc). During preparation of the exercises, we save the class definitions of the provided classes in interface view. When students want to find out about the services a class offers, they just double-click on the class symbol and the interface view is presented. Later, students start to extend the given class definitions and switch to the implementation view then.

While interactively exploring and extending the services of provided classes in the first weeks, students get an intuitive feeling for the *interface* of a class early on. During this time we do not make the distinction between methods and operations, as this would hinder more than help the learning process. The vocabulary in use thus contains primarily the words class, object, instance, method, and field. The field types of the classes in use are primitive types only in the first four weeks.

In week 5 we explicitly introduce the concept of reference types. We set reference types in relation to the primitive types and explain the notion of a general type as described in section 3.3, primarily to explain static type checking in Java.

4.2 `Interfaces` Next

In week 7 we introduce `interfaces` together with unit testing. We explain that a black box test should just test the *interface* of a class. We then show how they can use the Java mechanism `interface` to explicitly describe the *interface* of a class. If a test class is just using this `interface` it can be expected that the test cases will treat the object at test really as a black box. In the test class, any variable should be declared of the interface type, not of the concrete class type. The object to test is passed in as a constructor parameter, even this parameter can be of the interface type. As BlueJ allows passing objects interactively, the object to test can be created interactively and passed interactively to a test class object.

This introduction of `interfaces` automatically leads to the distinction of the static and the dynamic type of a variable. The static type of the variables should be of the `interface` type, the dynamic type is the concrete class type. This way, students consume dynamic binding without the necessity to understand the mechanics behind the scenes.

After writing the black box test class, we introduce JUnit as a framework for better test support. Using JUnit, students also consume inheritance (they passively inherit methods of the JUnit class `TestCase` that they can use for assertions), but they need not produce an inheritance relationship (provide an abstraction that can be reused).

4.3 Collection `Interfaces`

The second half of the semester is arranged around a central theme: collections of objects. Again we take a different approach than most text books: we introduce two basic collection `interfaces` of the Java Collection Framework (JCF), namely `Set` and `List`, before we introduce arrays (see [20] for supporting arguments for this approach). We explain the `interfaces` and let the students consume these types (and their implementations) to solve simple tasks with collections. At this point we explicitly do not talk about the type hierarchy of the JCF. As mentioned in section 3.5, students have to consume genericity without a full introduction of the concept.

For the rest of the semester we "open the hood" of the collection abstractions and show how lists can be implemented with linked lists and "growing" arrays, and how sets can be implemented efficiently with hashing. In the exercises, the students have to build these implementations for simple (non-generic) `interfaces` such as `ShortList` (for short lists in the sense of Nick Hornby's "High Fidelity") and `Vocabulary` (a set of strings for analyzing the corpus of a digital text). Arrays are described as a (very necessary) low-level construct only that is used to implement more user-friendly collections.

In summary, we use collection types as prime examples for *type abstraction* (section 5.3), without mentioning the term explicitly.

4.4 Students Feedback

SD1 was very well received by the students. In the "official" informal surveys conducted by the student union, the module got excellent marks, especially for its clear structure (over 80% voted "very good" or "good"). Students were quite aware of the fact that a different path was being taken with respect to `interfaces`, some were even excited about the novel approach. No student complained about the fact that inheritance was not taught and no student had problems with using interfaces the way we proposed.

5. ADVANCED TERMINOLOGY

The inheritance mechanisms in programming languages support several inheritance concepts. For first year teaching, the most important concepts are:

- subtyping
- (implementation) inheritance
- type abstraction

Basically, the first two concepts subsume hierarchies at the type level and at the implementation level, respectively, whereas type abstraction covers the relation between them.

5.1 Subtyping

Subtyping allows one to build hierarchies of types that support the notion of *substitutability* – (instances of) subtypes are allowed where supertypes are expected. Substitutability requires *dynamic binding*, as the compiler cannot decide at compile-time which method should be chosen for the invocation of an operation.

Subtyping is a very strict concept. It puts certain restrictions on the way inherited operations can be changed in a subtype. The folklore results from object-oriented type theory are that result types of operations can be adapted covariantly and that parameter types can be adapted contravariantly. In connection with genericity, subtyping rules tend to become bulky.

5.2 Inheritance

According to [22] and [19], *inheritance* is an incremental modification mechanism. This applies primarily to code, as the modification of signatures (e.g. covariant changes of parameter types) is very restricted, at least as long as substitutability is desired.

The most important underlying concept for implementation inheritance is *method redefinition*: the ability of a subclass to adapt inherited methods. We distinguish three forms:

- method *definition*: providing a concrete method for an abstract method.
- method *replacement*: overriding an inherited method without calling it from the new method body.
- method *extension*: overriding an inherited method, but calling it from the new method body.

A prerequisite for method redefinition is some kind of *late-bound self-reference*: Inside a class hierarchy, older code in a superclass can call newer code from a subclass, because calls can be targeted to the special object reference `this`. This seems to be quite similar to dynamic binding, but there is an important difference between late-bound self-reference and substitutability: If the source code of a superclass is available, late-bound self-reference can be resolved at compile time, simply by copying. This solution was chosen, for example, for Sather [18]. But the most common solution is indeed to implement late-bound self-reference with dynamic binding as well, and this is also the case in Java.

5.3 Type Abstraction

With *type abstraction* we denote the concept that an abstract data type can be implemented in several ways. The abstract data type *list* (user-defined order, duplicates allowed) can be implemented as a linked list or with growing arrays (see the classes `LinkedList` and `ArrayList` in the JCF). Correct implementations only differ in their runtime efficiency, not in their semantics. The abstract data type *set* (no visible order, duplicates not allowed) can be implemented with a simple linked list or, more sophisticated, with hashing.

We took the term type abstraction from [3], but the concept is also known as *data abstraction*, as described by Liskov in [13]. The latter paper is also the reason why we distinguish type abstraction from type hierarchies: the paper argues quite convincingly that the "implements" relation should be distinguished from the "is a" relation.

6. TEACHING SUBTYPING BEFORE INHERITANCE

As SD1 is a prerequisite for SD2, we expect that the distinction between interface/type and implementation/class is well received by all students at the beginning of SD2.

We believe that knowledge about the distinction and the differences between subtyping and inheritance is important for any well-educated software engineer. But the problem with most object-oriented programming languages is that classes are both types and implementations and that the inheritance relationship between classes typically implies both subtyping and code inheritance. So we cover the two topics in strict sequence and try to make the differences as clear as possible. We expect most of the students never to encounter such a sharp distinction again in their professional life.

6.1 Teaching Subtyping

This leaves the question of what to cover first. We decided to start with subtyping, as this seems to be the natural next step after the introduction of interfaces in SD1. If interfaces can be used to model abstract data types, then modeling a type hierarchy for substitutability is straightforward. One example that we discuss in the lecture is the `interface Collection` as a supertype for the already known types `List` and `Set`.

In the exercises, students then have to model a database of media items with CDs, DVDs and video games. Provided is an interface `Medium` that models the general properties of a multimedia item in a video store. When students implement the classes `CD`, `DVD` and `VideoGame` no inheritance of code is involved. We further provide a class `AbstractMedium` that implements the handling of properties common to all media and let the students use this class via forwarding (aka delegation).

In the following week, students then have to implement generic versions of their collection implementations from SD1. Both genericity and type hierarchies in Java are concepts on the type level. Up to this point, students still had no explicit contact with method redefinition (technically, they consumed method definition by implementing abstract methods from interfaces, but we avoid this terminology to this point). In the lecture we focus on the several features of types: subtyping, co- and contravariance (both for method parameters and for type parameters in generics), constrained genericity.

6.2 Teaching Inheritance

Only in the following week, we finally reveal everything about inheritance in Java in the lecture: we introduce the different types of method redefinition, late-bound self-reference and the concept of template methods; we discuss the keyword `protected` and the resulting heir *interface*. In the exercises the students then have to change their delegation implementation of the media database into an inheriting one, as we provide a new class

`AbstractMedium` with a template method. This method calculates the price depending on a hook method the subclasses have to implement.

The next week is dedicated to graphical user interfaces. In the introduction of AWT and Swing, the students need to have a sound understanding of both subtyping and inheritance.

6.3 Students Feedback

At the end of SE2, we conducted a survey in which the students could select between given answers to statements about the way we taught `interfaces`, subtyping and inheritance in SD1 and SD2. In the answers, 90% supported the statement that inheritance concepts are a crucial aspect of object-oriented programming. 62% fully supported that the distinction between subtyping and inheritance should be taught, 38% voted at least for "weak support". 86% preferred the presented order of topics, whereas 14% would have preferred inheritance to be covered before subtyping. 90% supported the statement that it is good design to cover interfaces in SD1 without a full explanation of inheritance.

7. CONCLUSION

In this paper, we presented our design rationale for two consecutive introductory courses on object-oriented programming. We focused on our novel way of introducing inheritance concepts: introducing type abstraction with interfaces (both the concept and the mechanism) before inheritance and explicitly separating subtyping from (implementation) inheritance. This approach is feasible only because Java offers a dedicated mechanism for modeling type information without any code burden: `interfaces`. We believe that `interfaces` in Java are a much more fundamental concept than their technical roots make them appear. They are, considered as fully abstract classes, not just some special case of the important notion of abstract classes; they allow to model the *interface* of a class explicitly and they allow to model multiple type hierarchies. An extensive use of `interfaces` typically improves the structure of object-oriented systems, whereas the extensive use of inheritance typically leads to code that is more difficult to maintain. Following the Early Bird pattern, any software engineering curriculum should try to cover `interfaces` as early as possible. If covered in isolation in semester 1, a smooth transition to inheritance via subtyping is possible in semester 2, as we pointed out in this paper.

8. REFERENCES

[1] The Pedagogical Patterns Project, http://www.pedagogicalpatterns.org, (last visited May 23, 2006).

[2] Barnes, D. and Kölling, M. *Objects First with Java - A Practical Introduction Using BlueJ (3rd Edition)*. Pearson Education, UK, 2006.

[3] Baumgartner, G. and Russo, V.F. Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++. *Software - Practice and Experience, 25* (8), 1995. 863-889.

[4] Bishop, J., Bishop, J.M. and Bishop, N. *Java Gently for Engineers and Scientists*. Addison Wesley, 2000.

[5] Cook, W., Hill, W. and Canning, P., Inheritance is Not Subtyping. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, (1990), 125-135.

[6] Evered, M., Keedy, J.L., Schmolitzky, A. and Menger, G., How Well Do Inheritance Mechanisms support Inheritance Concepts? In *Proc. Joint Modular Languages Conference (JMLC) '97*, (Linz, Austria, 1997), Lecture Notes in Computer Science 1204, 252-266.

[7] Halbert, D.C. and O'Brien, P.D., Using Types and Inheritance in Object-Oriented Languages. In *Proc. ECOOP '87*, (Paris, France, 1987), Lecture Notes in Computer Science 276, 20-31.

[8] Hoare, C.A.R. Proofs of Correctness of Data Representation. *Acta Informatica, 1* (4), 1972. 271-281.

[9] Horstmann, C.S. *Object-Oriented Design and Patterns*. John Wiley & Sons, 2006.

[10] Jia, X. *Object Oriented Software Development using Java*. Addison Wesley, 2002.

[11] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, 13* (4), 2003. 249-268.

[12] LaLonde, W. and Pugh, J. Subclassing != subtyping != Is-a. *Journal of Object-oriented Programming* (January), 1991. 57-62.

[13] Liskov, B., Data Abstraction and Hierarchy. In *Proc. OOPSLA '87 (Addendum)*, (Orlando, Florida, 1988), ACM SIGPLAN Notices.

[14] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall, New York, 1988.

[15] Porter, I., H. H. Separating the Subtype Hierarchy from the Inheritance of Implementation *Journal of Object-Oriented Programming*, 1992, 28-34.

[16] Schmolitzky, A., "Objects First, Interfaces Next" or Interfaces Before Inheritance. In *Proc. OOPSLA '04 (Companion: Educators' Symposium)*, (Vancouver, BC, Canada, 2004), ACM Press.

[17] Steimann, F., Siberski, W. and Kühne, T., Towards the Systematic Use of Interfaces in Java Programming. In *Proc. Proc. of the 2nd Int. Conf. on the Principles and Practice of Programming in Java PPPJ*, (Kilkenny, Ireland, 2003), 13-17.

[18] Szypersky, C., Omohundro, S. and Murer, S. Engineering a Programming Language: The Type and Class System of Sather. in Gutknecht, J. ed. *Programming Languages and System Architectures*, Springer-Verlag, 1993, 208-227.

[19] Taivalsaari, A. On the Notion of Inheritance. *ACM Computing Surveys*, *28* (3), 1996. 438-479.

[20] Ventura, P., Egert, C. and Decker, A., Ancestor worship in {CS1}: on the primacy of arrays. In *Proc. OOPSLA '04 (Companion: Educators' Symposium)*, (Vancouver, BC, Canada, 2004), ACM Press.

[21] Wampler, B.E. *The Essence of Object-Oriented Programming with Java and UML*. Addison Wesley, 2001.

[22] Wegner, P. and Zdonik, S.B., Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proc. ECOOP '88*, (Oslo, Norway, 1988), Lecture Notes in Computer Science 322.

Improving the Quality of Programming Education by Online Assessment

Gregor Fischer
University of Würzburg
Am Hubland
97074 Würzburg
+49 / 931 / 888 – 66 11
fischer@informatik.uni-wuerzburg.de

Jürgen Wolff von Gudenberg
University of Würzburg
Am Hubland
97074 Würzburg
+49 / 931 / 888 – 66 02
wolff@informatik.uni-wuerzburg.de

ABSTRACT

The paper presents an online Java course consisting of a tutorial that provides a high level of interaction and an assessment tool that analyses the code and enables the students to run a suite of predefined tests.

The hypertext tutorial contains a lot of interactive, editable examples and many exercises to check the student's progress.

In the assessment tool various electronic evaluators check for the conformance of uploaded student programs to coding conventions, proper documentation, compliance with the specification and, last but not least, the correct execution of supplied functional tests.

The tool not only provides a tremendous help for the correctors by reducing the manual assessment time by a factor of about 4, but also is appreciated by the students for its immediate reaction, because development times (especially during debugging) can be shortened considerably and students can gain a much higher confidence in the quality of their own program.

Categories and Subject Descriptors

D.2.5 **[Software Engineering]**: Testing and Debugging – *Code inspections and walk-throughs, Testing tools.*

K.3.1 **[Computers and Education]**: Computer Uses in Education – *Computer-assisted instruction (CAI), Computer-managed instruction (CMI), Distance learning.*

K.3.2 **[Computers and Education]**: Computer and Information Science Education – *Computer science education.*

General Terms

Measurement, Reliability.

Keywords

Assessment, Programming Education, Quality of Programs, Teaching, Testing.

1. OBJECTIVES

A sound and thorough education in programming is one of the key qualifications a student of computer science has to have and it is further a mandatory requirement for most jobs in IT, networking, etc. A clear and pleasing representation of the concepts of a programming paradigm and its realisation in a specific language as well as the opportunity for many practical exercises are crucial for the success of a course [3].

Since more than 25 years, we have made a lot of experience in running programming courses for different languages and paradigms. The lectures always included assignments for programming exercises, which were mostly handed in as handwritten programs via e-mail or WWW, in former times even as print-outs. These solutions were then manually inspected and evaluated.

It is our educational objective to advice the students to write good programs and not only to learn the syntax of the language. Programming is a creative process, it can only be learned by reading and, first of all, writing programs. In a more detailed study in [1] five steps of learning are distinguished: following, coding, understanding and integrating, problem solving, and participating. In this terminology we emphasize the first 3 layers.

For all these steps we provide an immediate response. All sample programs are executable and can be modified easily, hence, the reading or following task is supported. For the coding step we do not only rely on the error messages generated by a compiler, but also enforce several coding and documenting conventions. Our main focus, however is the provision of immediate, automatic checks for the written program. Hence the students learn to understand the meaning of the language constructs. The instantaneous feedback increases the motivation quite considerably.

Assignments of programming courses, hence, have to cover more than the syntax of the used language. We want to see and assess complete programs. Multiple choice questions are not appropriate, since they turn to focus on very particular or peculiar properties instead of supporting a good, solid programming style. That can be achieved by formal or structural tests (see below). The proper functionality of a given program may be assessed by checking its output (e.g. as a free form text). Again care must be taken in order to avoid problems with irrelevant details like whitespaces or formatting.

More sophisticated tests are necessary to judge the quality and functionality of uploaded programming exercises, such as the style of writing and formatting and the correctness of results. In traditional courses these properties should be - and were actually - assessed by looking into the source code and documentation of the student's solution. Hence, the assessment of programming assignments usually took a lot of human corrector time.

The automation of these reviews and quality assurance tests by our tool is described in the following section.

2. AUTOMATIC ASSESSMENT OF JAVA PROGRAMS

In recent years we have developed an online Java course consisting of a tutorial that provides a high level of interaction and an assessment tool that analyses the code and enables the students to run a suite of predefined tests [2].

The hypertext tutorial can be navigated via a predefined learning path or by freely following the hyperlinks. It contains a lot of editable and executable examples and many exercises (from multiple choice and free text to programming exercises) to check the student's progress.

The solutions for the programming exercises, small or medium-sized programs, are uploaded to the assessment tool [4]. Thence, various electronic evaluators perform:

- *Formal tests*
  We check if the program compiles, conforms to coding and naming conventions, and if appropriate comments are present such as Javadoc comments.
- *Structural tests*
  The recommended usage of structured statements or data types is checked. Violation of these conventions may lead to hidden mistakes. It is e.g. possible to enforce that the statement following a loop or branch statement is a block or that case clauses are terminated by a break or a return statement.
- *Specification tests*
  The specification requires that classes with predefined names or methods with given signatures are expected. Classes have to extend other classes or implement given interfaces. In a teaching environment it may be interesting to prohibit the usage of packages or classes.
- *Functional tests*
  Comparison with a master solution or a set of JUnit tests are performed.

2.1 Functional Tests

The common practice for functional tests is black box testing, using e.g. the JUnit framework. These tests check the solution for well known test cases and their expected results. Therefore these results must be known in advance, what might be problematic, if the way to compute the results is rather complex.

Whereas the development of black box tests is or at least should be standard in a development team we also provide the opportunity to compare the results of the student solution with a master solution.

The comparator provides a set of classes and methods that mirrors the structure of both solutions. The implementation then calls the respective constructs from the student's and the master's implementation, and compares the results. The comparison is done on the real objects, taking into account the internal structure of arrays and collections. Special methods for comparison can be provided such as a possible equality relation for floating-point numbers that only checks, is the student's solution lies within a given interval.

This simplifies the writing of tests tremendously, as (seemingly) standard manipulation of objects can be used, and no explicit checking needs to be done. Of course explicit checking is also possible, in case the implicit is not sufficient.

2.2 Combining Tests

The tests can be configured and individually tuned for each specific exercise. The combination of different tests makes sense especially in a teaching environment where we are interested in more than results. We also want to know or prescribe the manner these results are produced. A method may have to be called recursively, another must not use a while loop and a third method has to advance an iterator of a data structure properly. The results have to be the same. Such conditions or requirements can easily be imposed by our tool by combining the appropriate (e.g. the structural and functional) tests.

All tests may be mandatory, optional or even secret. A program is only accepted, if it passes the required checks. Since the automatic assessment is carried out immediately, the students do not need to wait for feedback and can fix problems and upload corrected versions immediately. The number of turn in tries usually is not limited, there is only a general submission deadline for an exercise.

Having optional tests is especially important in courses, where programming is not the major objective. Here compliance to e.g. formal tests is often not enforced in order to allow students to concentrate on different matters. Often even some or all functional tests are configured as optional, so that partially incomplete solutions can be turned in, because rejecting those completely would be too strict. Yet, those tests are usually included, so that students remember, that their programs can still be improved.

3. EXAMPLE

Let us illustrate the work of our tool by a simple exercise:

Write a class StringReverse containing a method reverse(String) that reverses a string without using the method reverse() from the class java.lang.StringBuffer.

In the following we mimic the upload and assessment of several attempts to solve this exercise.

1.Attempt:

Uploaded program (shortened):

```
public static String reverse(String s) {
    return new StringBuffer(s).reverse().toString();
}
public static void main(String[] args) {
    for (int i=0; i<args.length; ++i) {
        System.out.println(reverse(args[i]));
    }
}
```

Results of online checks (shortened, translated from German):

```
Check 1: compilation: passed
Check 2: coding conventions: passed
Check 3: signature and hierarchy: failed
    Invocation test checks whether prohibited classes
    or methods are used; call of method reverse from
    the prohibited class java.lang.StringBuffer
```

2. Attempt:

Uploaded program (shortened):

```
public static String reverse(String s) {
    if (s == null) return null;
    if (s.length() <= 1) return s;
    char first = s.charAt(1);
    String rest = s.substring(1);
    return reverse(rest) + first;
}
```

Results of online checks (shortened, translated from German):

```
Check 1: compilation : passed
Check 2: coding conventions: passed
Check 3: signature and hierarchy: passed
Check 4: dynamic grey box test: failed
    StringReverse.reverse(java.lang.String)
    yields „llebe“ instead of „leben“
```

3. Attempt:

Uploaded program (shortened):

```
public static String reverse(String s) {
    if (s == null) return null;
    if (s.length() <= 1) return s;
    char first = s.charAt(0);
    String rest = s.substring(1);
    return reverse(rest) + first;
}
```

Results of online checks (shortened, translated from German):

```
Check 1: compilation: passed
Check 2: coding conventions: passed
Check 3: signature and hierarchy: passed
Check 4: dynamic grey box test: passed
```

4. TWO VERSIONS

Currently there are two versions (note that both are in German) of the tool available. A server based variant (at http://jop.informatik.uni-wuerzburg.de/), hosted by the University of Würzburg on behalf of the Virtual University of Bavaria (vhb). A demo version of the tutorial including assessment of exercises is available at http://jop.informatik.uni-wuerzburg.de/tutorial/demo/. The system is called Praktomat [6], it organizes the download of exercises and upload of solutions, it controls submission date and time, automatically runs the prepared checkers, manages the manual reviews, and supports e-mail communication with the students.

The second version is published as a book [5] and a CD, including the newly developed evaluation tool Java Exercise Evaluation Environment. JEEE allows editing, running and testing of examples and exercises in a secure and interactive environment, embedded in an easy to use graphical client. It integrates formal, structural, specification and functional tests, and can therefore be used as a replacement for the server based testing in offline use. It can also be used to further support human correctors, as it allows for an interactive evaluation of even modified student solutions. Nevertheless the server based system is still required at least for turn in and other management purposes.

Due to the high level of interactivity and automatic responses we expect for the new version that the support of the human correctors will be increased. Since it can be used without an internet connection, the flexibility of the students will be enlarged. This is particularly important in continuing education that is one of the main goals of the second version. We are convinced that the coaching time, i.e. the time of personal presence of the trainer can be reduced by a factor of at least 2. Individual schedules for each participant will be possible.

5. EVALUATION

The system has been used in various different lectures (“Algorithms and Datastructures”, “Software Engineering”, “Introduction to Computer Science for non CS Majors”, “Advanced Training for Teachers”, …) or practical courses (“Programming in Java” and “Software Development”) at the University of Würzburg or for the Virtual University of Bavaria (vhb). During these courses a total of more then 1000 subscribed students handed in more then 3000 programs (solutions) with an average size of about 1000 lines, which passed the required tests for over 80 different problems. More then one hundred thousand attempts were rejected because they did not fulfil all requirements. The high number of attempts can be explained by the fact that the students use the online assessment as convenient test tool during program development.

Table 1. Application of Online Assessment

Category	Number of Courses	Total Number of Students	Total Number of Attempts	Total Number of Solutions
Programming Lab	6	870	~121 000	2 933
Online Course	4	150	~13 000	442
Lectures for CS Majors	3	~350	~7 300	-
Lectures for non CS Majors	7	~500	~15 300	-

The tests and the immediate responses of the system helped the students very much. Indeed, they made it possible to cope with more advanced problems. Test configuration varied from only enforcing the solutions to be compilable to requiring full compliance with coding standards and an extensive functional test suite.

The reliability of the tests is proven by the fact that less than 2% of the programs that passed all tests were not accepted by a human inspector. Despite the higher quality the rate of successful participants has not changed in comparison with former courses.

Table 2. Programming lab breakdown
(note that different assessment policies were used)

	Spring 2006 (Programming lab)	Fall 2005/06 (Programming lab)	Spring 2005 (Programming lab)	Fall 2004/05 (Online course)
Enrolled Students	145	32	195	74
Participating Students	106 (73%)	24 (75%)	140 (72%)	25 (34%)
Successful Students	55 (52%)	17 (71%)	57 (41%)	13 (52%)
Exercises	16	15	11	8
Attempts	15 970	11 511	33 185	3 261
Solutions	1015	301	990	105
Excellent Solutions	41.2%	81.8%	64.5%	48.2%
Average Solutions	53.4%	15.9%	32.8%	49.5%
Bad Solutions	4.9%	0.5%	1.5%	1.4%
Manually Rejected Solutions	0.5%	1.8%	1.3%	0.9%

The readability and the quality of the accepted programs have increased tremendously. The solution of more comprehensive and more complicated exercises can be required. This goes hand in hand with the reduction of the assessment time for a human corrector by a factor of 3 to 4. A more thorough code inspection has been made possible. Of course, this final inspection is necessary, because we cannot proof correctness of programs online.

6. FUTURE WORK & CONCLUSION

While other systems like Maven [7] or CruiseControl [8] can also perform automatic program checking, they do not integrate well in a teaching environment, where students usually do not work as a team. They also lack the necessary resource constraint checks along with review- and grading support.

We therefore consider our system as necessary for our purpose. Because of architectural limitations in the current implementation a complete rewrite of the system is underway that focuses on:

- Service Oriented Architecture
- Maintainability
- Internationalization
- Tight integration with IDE
- Faster feedback times
- Improved support for correctors and advisors
- Cooperative and competitive programming

Yet, even with the current system we have shown that automatic assessment of programming exercises is possible. The supplied structural and functional test can reach a level of confidence that is high enough to reject false solutions. For the acceptance of seemingly correct solutions the precision is very high (more than 98%). For two reasons, however, we did not stop the human supervision, first it is an administrative legal problem, and second many programs that passed the tests could still be optimized and corresponding advice was given by the program inspectors. Altogether for programming courses we have achieved a higher quality of programs hand in hand with a more reliable and faster assessment.

7. REFERENCES

[1] Bruce et al: *Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University.* Journal of Information Technology Education Volume 3, 2004.

[2] H. Eichelberger et al: *Programmierausbildung Online.* DeLFI 2003, A.Bode,J.Desel,S.Rathmayer,M.Wessner (eds), Lecture Notes in Informatics, GI, p.134-143.

[3] G. Fischer, J. Wolff v. Gudenberg: *Java Online Pedagogy.* ECOOP'03 Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts.

[4] G. Fischer, J. Wolff v. Gudenberg: *Online Assessment of Programming Exercises.* ECOOP 2004 Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts.

[5] G. Fischer, J. Wolff von Gudenberg: *Programmieren in Java 1.5.* Springer-Verlag, Berlin, 2005.

[6] J. Krinke, M. Störzer, A. Zeller: *Web-basierte Programmierpraktika mit Praktomat.* Softwaretechnik-Trends 22:3, S.51-53, 2002.

[7] Maven - A software project management and comprehension tool. `http://maven.apache.org/`

[8] CruiseControl - A framework for a continuous build process. `http://cruisecontrol.sourceforge.net/`

Invited Workshop on

Java-based Distributed Systems and Middleware

Mannheim, Germany. August 30, 2006

Edited by: Axel Korthaus

Message from the Workshop Organizer

Most of today's complex software systems, e.g. enterprise, internet, peer-to-peer, grid, or mobile applications, are distributed in nature. In distributed architectures, middleware software layers are typically used to facilitate the separation of business logic and infrastructure components. Aspects such as diverse platforms, remote communication over networks, concurrency, distribution, persistence, and others make the development, deployment and monitoring of distributed systems rather difficult.

Java technology has become one of the predominant platforms for such applications. Numerous Java APIs exist to provide a secure, robust, and scalable environment for building, deploying, and managing distributed applications while delivering a high level of code portability and reusability. However, researchers and developers alike still contend with questions regarding how to build Java-based distributed systems effectively and efficiently and how to improve existing Java technologies for that purpose.

This workshop aims at providing a forum for the exchange of know-how on all aspects of the architecture, design, implementation, and management of Java-based distributed applications and middleware layers using the latest Java technologies and APIs.

The objectives of the workshop are to:

- provide a forum for the exchange of research ideas and results in the area of Java-based distributed systems and middleware;
- bring together experts from academia and industry;
- disseminate recent developments and stimulate exchange on future challenges;
- provide an overview of the current state-of-the-art and work-in-progress.

The workshop covers a wide range of different aspects of the general topic. In the first contribution, Gitzel and Schwind address the problem of generating code for J2EE-based Web applications using the OMEGA approach, a variant of Model-Driven Development that builds on the concepts promoted by Executable UML. The second paper, authored by Barolli, focuses on the domain of Java-based Peer-to-Peer Systems built on the JXTA technology. The proposed system, M3PS, is a multi-platform P2P system with a pure P2P architecture. The last two contributions take a SOA- and business process-centric view and describe experience with J2EE-based middleware gained in industrial settings at the Deutsche Post AG (Jobst and Preissler) and the norisbank AG (Greiner, Düster, Pouatcha, v. Ammon, Brandl, and Guschakowski). Business process management, business activity monitoring, SOA monitoring and management, and (complex) event processing are the predominant objects of investigation in those reports. Thus, the papers cover the dimensions of both research and practical experience. I hope that the workshop will provide an interesting cross-section of the subject matter.

I would like to thank the authors of the workshop submissions for their valuable contributions and interest in the workshop. Last but not least, I would like to thank Holger Bär (CAS Software AG), Stefan Kuhlins (UAS Heilbronn, Germany), Karsten Meinders (Q-Labs GmbH), Mark-Oliver Reiser (TU Berlin, Germany), and Jan Wiesenberger (FZI Karlsruhe, Germany), the members of the workshop's program committee, for their professional reviews and overall support.

Dr. Axel Korthaus
(Workshop Organizer)
University of Mannheim
Department of Information Systems
Schloss, L 5,5
68131 Mannheim (Germany)
Fax: +49 (621) 181-1643
Email: korthaus@uni-mannheim.de

Experiences With Hierarchy-Based Code Generation in the J2EE Context

Ralf Gitzel
University of Mannheim
Schloss
68131 Mannheim, Germany
+49(0)621/181-1645
gitzel@wifo3.uni-mannheim.de

Michael Schwind
University of Mannheim
Schloss
68131 Mannheim, Germany
+49(0)621/181-1622
schwind@wifo3.uni-mannheim.de

ABSTRACT

OMEGA is a model-driven code generation approach based on Executable UML enhanced with extension information provided by metamodel hierarchies. It is domain-specific but can easily incorporate new or related domains. In this experience report, we describe the challenges encountered during the implementation of a code-generation facility for the J2EE platform.

Categories and Subject Descriptors

D 2.2 [**Design Tools and Techniques**]: Computer-Aided Software Engineering (CASE), State Diagrams – *Model-Driven Development, Domain-Specific Modeling, OMEGA*

D 2.6 [**Programming Environments**]: Graphical Environments – *Eclipse Plugin*

General Terms

Design

Keywords

Code Generation, Model-Driven Development, MDD, Executable UML

1. INTRODUCTION

The goal of this paper is the description of our experience with designing and implementing a Java code generation facility for the OMEGA modeling approach. The emphasis of the paper is on the description of the code generation process and the Java-related challenges we have faced during the generator's implementation. Therefore, we provide a high-level simplified overview on the hierarchical nature of the approach and provide pointers to more detailed descriptions where appropriate. Our target audience are practitioners and researchers interested in the generation of runnable systems from class and state chart diagrams in a translationist approach (cf. [1]).

The paper is structured as follows: after a brief introduction to the basic principles of model-driven development (MDD), we introduce the hierarchical OMEGA approach and present those of its features that are relevant for the further discussion of the code generation process and the Java-related issues that we have experienced during the creation of the code generator. The presentation of the OMEGA approach is followed by a description of the steps necessary to transform a set of static and dynamic OMEGA models into executable Java code. The paper concludes with an overview of selected problems we had to solve during the implementation of the code generator and a summary of the experience we have gained in the process.

2. MODEL-DRIVEN DEVELOPMENT

Model-Driven Development (MDD), the successor to Computer Aided Software Engineering (CASE), is a relatively new idea aimed at putting abstract models at the center of the development effort, making them the most important artifacts for the whole software lifecycle. With the help of code generators applied to these models, software engineering takes a step away from programming. The focus on models is meant to increase the level of abstraction involved in the creation and alteration of software and to reduce a product's vulnerability to change during its whole lifecycle.

Several factors distinguish MDD from its predecessors. First of all, MDD is based on standards such as XMI, MOF and UML, reducing the effort required for infrastructural developments [2]. Also, without such standards, there is a risk when creating a model or visual program, because a proprietary solution might not be supported in the long run ([3], pg. 118, [4], and [2]). Furthermore, MDD is deliberately kept free from ties to a specific design methodology ([3], pg. 118). Like compilers, MDD tools should be usable in many different contexts instead of constraining its users to a specific, possibly sub-optimal design method. Without these two factors, MDD would be little more than a renaissance of past concepts.

The most well-known MDD outgrowth is the OMG's Model-Driven Architecture (MDA) [5]. However, there are also MDD concepts such as Executable UML [6] that are largely unrelated to MDA. For a good discussion of the essentials of MDD see [2], [4], or – for a more critical view – [7].

3. THE OMEGA APPROACH

The MDD approach presented in this paper is based on the Ontological Metamodel Extension for Generative Architectures (OMEGA), a non-linear extension to the MOF metamodel (cf. [8]). In principle, the OMEGA approach is similar to Executable

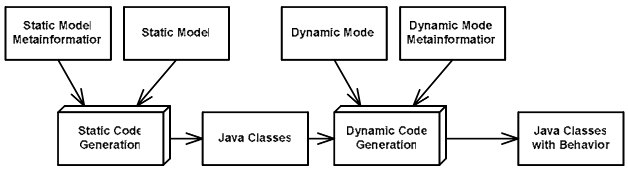

Static Model Metainformatior
Static Model
Dynamic Mode
Dynamic Mode Metainformatior
Static Code Generation
Java Classes
Dynamic Code Generation
Java Classes with Behavior

Figure 1 - OMEGA Principle

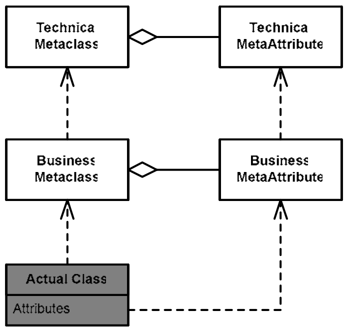


Figure 2 - Static Metaelements

UML [6]; class and statechart diagrams are used to model a system and serve as input for a code generator.

Figure 3 shows a simplified conceptual presentation of the OMEGA code generation process. OMEGA extends Executable UML by introducing a hierarchical extension mechanism that holds more meaning than UML Stereotypes ([9], pp. 164-178) and encodes domain-specific extensions but is in principle "backwards compatible" to the Stereotype concept. Effectively, each element in the model (static or dynamic) has one or more metaelements, which convey additional information useful for code generation. All information about the static models (class diagrams) is used to generate the basic classes of the final program and all information about the behavior, presented by statecharts, is used to add the code responsible for the dynamic aspects. It is not possible to give all the details of the approach in this paper, especially the formal aspects of the metamodel, for which we refer you to [10]. Instead, we provide a very usage-oriented description of the theory and focus on the description of the Java code generation issues in section 6.

3.1 General Structure

The hierarchical extension mechanism of OMEGA is realized as a multi-level hierarchy, giving each element up to two additional levels of abstraction using metaelements. Thus, each element is categorized twice based both on technical and on business aspects. In this paper's examples, we will use Web Applications as technical domain and Content Management Systems as business domain.

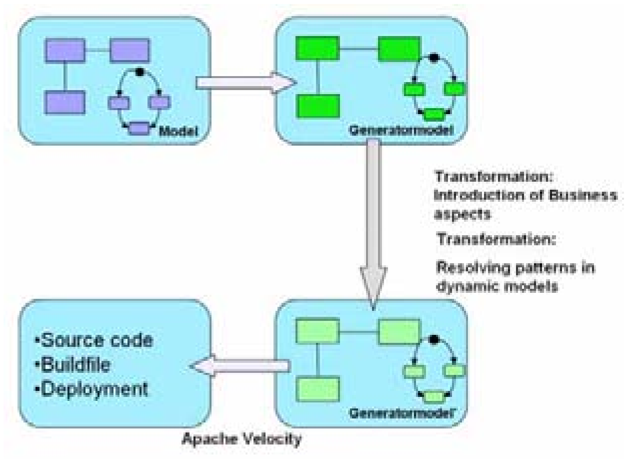


Figure 3 - Overview of the code generation process

The different static and dynamic metaelements are described in sections 3.2 and 3.3. The full example hierarchy is described in section 4.

3.2 Metaelements in Static Models

The most important metaelements in the static models are the metaclasses and the metaattributes (see Figure 2). Typically, a class (shown in dark grey) will have two metaclasses and each attribute up to two metaattributes. The metaclass determines which kinds of attributes a class may have by providing a list of metaattributes. The class, which is instance of the metaclass may only contain attributes that are instances of the metaattributes. The metaattributes in turn limit the set of allowed attribute types, visibilities, and multiciplicities for their attribute.

As an example, consider a class *ScientificPaper* as shown in Figure 10. It is an instance of the business metaclass *Article* (in the domain of content management systems as shown in Figure 6), which in turn is an instance of the technical metaclass *DataSet* (of the Web application domain, Figure 5).

Without any metadata, a code generation tool has no information about the concept of *ScientificPaper*. However, using the information provided by the metaclasses, the tool can combine business-specific and technology-specific knowledge to generate appropriate code. For example, as an instance of DataSet, ScientificArticle requires a persistence mechanism and as an instance of Article, it requires code for processes such as reviews or editorial change. An attribute of *ScientificPaper* called *Time* can be identified as the primary key of an entry if it is an instance of the technical metaattribute *Key* and as describing the time when the paper was submitted if known to be an instance of the business metaattribute *Timestamp*.

The associations in the class diagram also have metaelements that serve a similar purpose. An OMEGA association consists of two ends, both of which contain information on the end's allowed class, multiplicity, navigability, and possibly aggregation type. Similarly to attributes in a class the values allowed for an association end are controlled by metaassociation ends. Again, the introduction of metaelements allows a code generator to reason about classes in the domain.

For example, the Class *ScientificPaper* introduced above might be connected to a class *LibraryView* representing a Web page for displaying the paper. *Papers*, the association defined for this purpose has a multiplicity of 0..n and a navigability of *true* on the *ScientificPaper* side, as defined by one of its association ends. These values are from the set of allowed values provided by the business metaassociation *ShowsArticle*, which allows multiplicities from the set {0..n, 1..n} and a navigability from the set {*true*}. These sets in turn are subsets of the allowed values given by the technical metaassociation *Data*, which allows its leaf

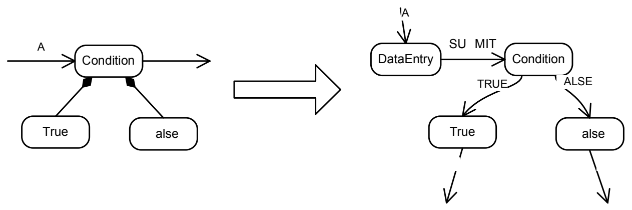


Figure 4 - Substate Replacement Pattern [10]

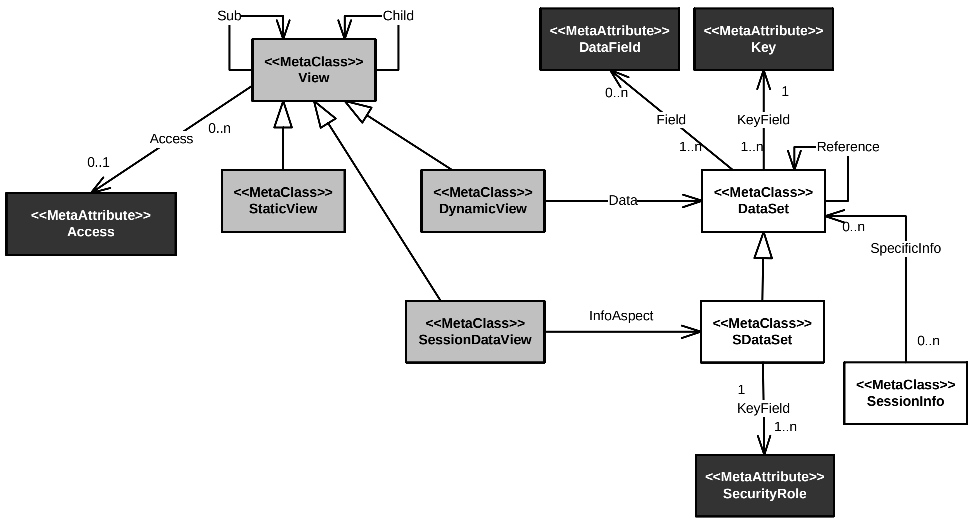


Figure 5 - The Static Technical Metamodel

instances to have a multiplicity from the set {0..n, 1..n, 1..1} and a navigability from the set {true}.

The principal benefit of OMEGA for code generation is that a lot of possible domain violations are automatically excluded. For example, only a navigability of *true* makes sense, since the Web page must be able to access the data it is supposed to display.

3.3 Metaelements in Dynamic Models

While the class diagram gives a good idea about the purpose of the application, dynamic models in the form of statechart diagrams are needed to adequately describe the application's behavior. As a subset of UML's statechart diagrams, the OMEGA dynamic diagrams allow the modeling of states, transitions, and entry code in the states. Again, these diagrams are extended with metaelements that describe typical domain-specific behavior.

Unlike static elements, states only have a single metaelement or even none at all. However, this metaelement may be from either the technical or the business domain. There are two different kinds of metastates on both layers: implicit and explicit metastates.

An implicit metastate does not require a definition of entry code. Instead, its behavior is *implicitly* defined by the metamodel. For example, a state called *SubmitPaper*, that is instance of the implicit metastate *CreateArticle*, would not need to contain any entry code, as the code generator understands the process of creating a new article based on user input (by first providing a HTML form with input fields for all attributes of type *Content* and assigning the current system time to all attributes of type *Timestamp*).

Explicit metastates, on the other hand, require their instances to contain entry code. These instances are used as parts of a pattern as shown on the left side of Figure 4, where *Condition* is an instance of an implicit metastate and *True* and *False* are instances of an explicit one. As shown on the right side, the states are fitted into a more complex state machine that automatically handles the condition and leads to the right code as defined in *True* or *False*, respectively.

4. CONTENT MANAGEMENT AS EXAMPLE DOMAIN

It has been stated already, that in this paper we use the technical domain of Web applications and the business domain of Content Management Systems as examples. We will briefly describe key aspects of the metamodels, which will be used to illustrate not only the code generation theory but also the challenges caused by the use of J2EE as output target.

The business metamodel (here: Content Management) elements are the direct metaelements of the elements in the application models and they are in turn instances of the technical metamodels (here: Web applications), influenced by their respective structures. Thus, we will start our discussion with a description of key elements from Figure 5 and Figure 7, which show the technical layer. Next, it will be explained how these are used to model the business domain, which is shown in Figure 6 and Figure 9. The business metamodels are the ones directly used for the actual models of the applications. Figure 10 and Figure 8 will serve as an example model to further illustrate the use of the hierarchy. In all diagrams, an element's type[1] (e.g., class or attribute) is shown in guillemets. Its metaelement from the technical or business domain[2] is shown in brackets.

4.1 General Structure

Figure 5 can be roughly divided into three kinds of elements. Those shown in light gray represent Web elements loosely based concepts introduced by OOHDM's Web modeling language (cf. [12]). Each *View* added to another one via the *Child* association will be part of that navigational context, creating the typical tree-like navigational structure favored by most sites.

The classes in white represent persistent data, their potential metaattributes are shown in dark gray. Instances of *DataSet* contain any kind of data such as customer information or virtual

[1] For those familiar with our work such as [10]: more precisely, this should be called the linguistic type

[2] More precisely: its ontological type

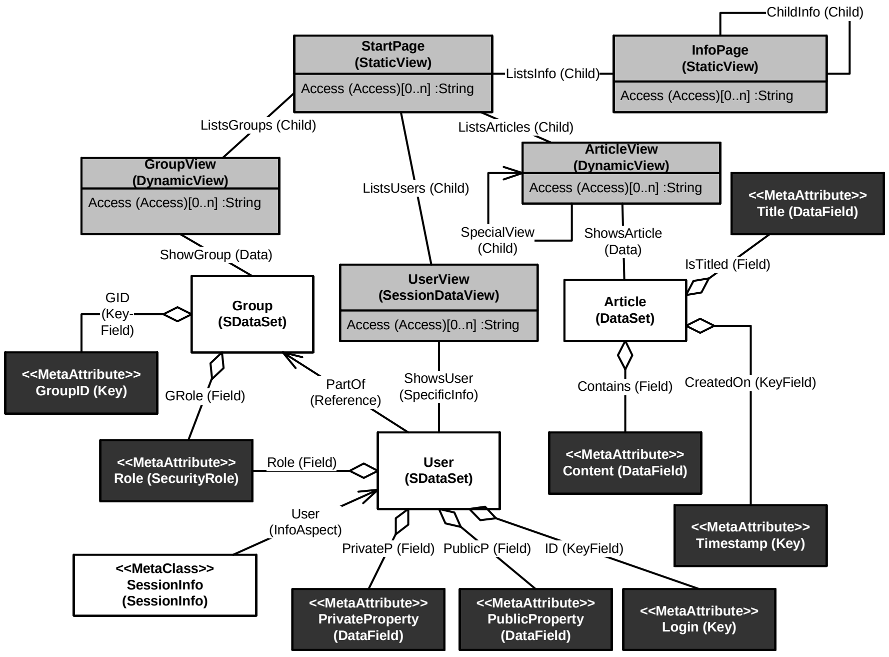

ChildInfo (Child)
StartPage (StaticView)
Access (Access)[0..n] :String
ListsInfo (Child)
InfoPage (StaticView)
Access (Access)[0..n] :String
ListsGroups (Child)
ListsArticles (Child)
GroupView (DynamicView)
Access (Access)[0..n] :String
ListsUsers (Child)
ArticleView (DynamicView)
Access (Access)[0..n] :String
<<MetaAttribute>> Title (DataField)
SpecialView (Child)
ShowsArticle (Data)
ShowGroup (Data)
IsTitled (Field)
UserView (SessionDataView)
Access (Access)[0..n] :String
GID (Key-Field)
Group (SDataSet)
Article (DataSet)
<<MetaAttribute>> GroupID (Key)
GRole (Field)
PartOf (Reference)
ShowsUser (SpecificInfo)
Contains (Field)
CreatedOn (KeyField)
<<MetaAttribute>> Role (SecurityRole)
Role (Field)
User (SDataSet)
<<MetaAttribute>> Content (DataField)
User (InfoAspect)
<<MetaAttribute>> Timestamp (Key)
PrivateP (Field)
PublicP (Field)
ID (KeyField)
<<MetaClass>> SessionInfo (SessionInfo)
<<MetaAttribute>> PrivateProperty (DataField)
<<MetaAttribute>> PublicProperty (DataField)
<<MetaAttribute>> Login (Key)

Figure 6 - Static Business Metamodel

products. A *DataSet* consists of a *Key*, allowing its unique identification, and several *DataFields*, describing its data. *SData* is used with the security mechanism and is explained in section 4.3.

The same division can be seen in Figure 6, which is the instance of Figure 5. We will use the example of the *DataSet*-instance *Article* to better explain this relationship in the next section.

Figure 7 shows the dynamic technical metamodel for Web applications. At the technical level there are only two kinds of statecharts (shown in gray), one for views on data and a special one to display and use session information. Both will be explained in the next two sections.

Figure 9 shows an excerpt of the dynamic business metamodel. Some of the statechart types here are extensions of the general types defined in Figure 7, others are new.

4.2 Articles and Reviews

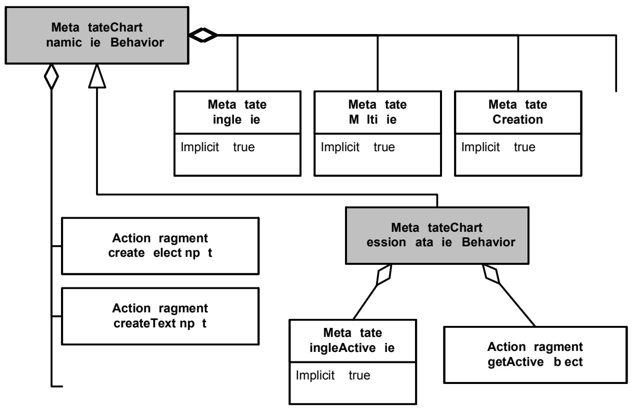

Meta tateChart namic ie Behavior
Meta tate ingle ie
Implicit true
Meta tate M lti ie
Implicit true
Meta tate Creation
Implicit true
Action ragment create elect np t
Action ragment createText np t
Meta tateChart ession ata ie Behavior
Meta tate ingleActive ie
Implicit true
Action ragment getActive b ect

Figure 7 - The Dynamic Technical Metamodel (Excerpt)

In order to better illustrate the J2EE code generation mechanism in section 5, a small selection of elements will be discussed in detail. For the purpose of illustration a closer look will be taken at the *Article* metaclass, the associated *ArticleView*, and their behavior.

The relevant metaclasses at the top of the static hierarchy (in the upper right quarter of Figure 5) are *DynamicView*, *DataSet*, *DataField*, and *Key*. The *DynamicView* represents a Web page or part of a Web page that displays the content of a *DataSet* or provides the user with means to alter its contents. The exact behavior depends on the associated statechart. *DynamicView* contains a single metaattribute called *Access* that is used for security purposes, defining which roles may access this web page.

The *DataSet* contains exactly one *Key* (due to the multiplicity of the association *Key*) and zero or more *DataFields*. The metaattributes *DataField* and *Key* define which kinds of attributes are allowed in instances of *DataSet*. A *DataField* allows the inclusion of attributes that have an arbitrary multiplicity and data type, the attribute allowed by *Key* must be of type *String*, *Double*, *Long*, or *int* and have a multiplicity of 1..1.

For a *DynamicView*'s associated statechart, several metastates exist at the technical level as shown in Figure 7. These include all kinds of typical behavior expected from a view on a data element. For example, *SingleView* shows an element with a specific key value and *MultiView* shows a list of all instances of the *DataSet* associated with the *DynamicView*. *Creation* presents a fill-out form, which allows the user to enter the data for a new *DataSet* instance.

The elements presented so far maintain a purely technical view on the system under study. The business level now adds the new perspective of the content management domain. Figure 6 contains *Article*, along with its metaattributes such as *Title* and *TimeStamp*, and *ArticleView*. All of these are instances of the afore-mentioned

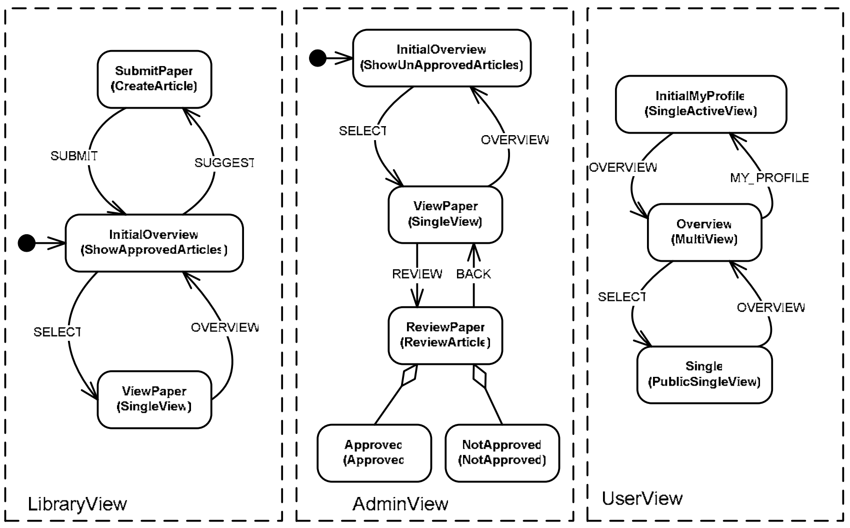

SubmitPaper (CreateArticle)
SUBMIT
SUGGEST
InitialOverview (ShowApprovedArticles)
SELECT
OVERVIEW
ViewPaper (SingleView)
LibraryView
InitialOverview (ShowUnApprovedArticles)
SELECT
OVERVIEW
ViewPaper (SingleView)
REVIEW
BACK
ReviewPaper (ReviewArticle)
Approved (Approved)
NotApproved (NotApproved)
AdminView
InitialMyProfile (SingleActiveView)
OVERVIEW
MY_PROFILE
Overview (MultiView)
SELECT
OVERVIEW
Single (PublicSingleView)
UserView

Figure 8 - Actual State Charts

elements of the technical view, as indicated by the names in brackets.

An *Article* is an instance of *DataSet* that adds additional semantics, i.e. is the specification of a document in a content management system that can be reviewed, accepted, or rejected. Similarly, the metaattributes are semantically enriched.

For example, *TimeStamp* has all the limitations of its metaelement *Key* but is further restricted, as only *Long* is allowed as data type. The attribute value is interpreted to be the date of submission. Thus, it is possible to automatically assign a key value during creation as there are concise rules on how to determine it. *Title* is an instance of *Content* that is limited to *String* values. When creating code for the application, the generator is able to use an instance of *Title* as the short form for the *Article*.

The behavior of an *ArticleView* and its *Article* is determined by all elements of *ArticleViewBehavior* and *ArticleBehavior* in Figure 9. The metastate *CreateArticle* for example is similar to *Creation* but has more knowledge about the nature of the data entered. It will automatically set the value for *TimeStamp* and can chose to provide a smaller text field for the *Title* than for the *Content*.

Of special interest are the metastate *ReviewArticle* and its "substates" *Approved* and *NotApproved*. Their instances form part of a pattern (see Figure 4). *ReviewArticle* is an implicit metastate. Depending on whether the *Article* is accepted or not, either the state marked as *Approved* or the one called *NotApproved* will be entered next. Both are explicit metastates, which means that they contain user-defined action code. Thus, the modeler is able to specify custom behavior for acceptance and rejection of submitted papers.

Figure 10 shows the actual static model for a digital library. There are two different *ArticleViews* called *LibraryView* and *AdminView*. Both are connected to the same *Article* type called *ScientificPaper* but have different statecharts and thus offer different functionality. *ScientificPaper* has several attributes, all of which conform to one of the metaattributes in *Article*.

The statechart associated with *AdminView* (see Figure 8, middle) provides a good example of how OMEGA handles dynamic aspects of a model. Each of the states has a metastate that is either technical (such as *SingleView*) or business-oriented (such as *ShowUnapprovedArticles*). Most of these states use implicit metastates and therefore do not require action code, automatically providing the functionality needed.

If "Approve" is chosen, the statechart changes to the state *Approved* and executes that state's entry code[3]:

```
Long key =
     (Long)loadFromSession("CurrentArticleKey");
ScientificPaper paper =
     Papers.findByPrimaryKey(key);
paper.setApproved(new Boolean(true));
Papers.update(paper);
```

Otherwise, the state *NotApproved* is entered and the entry code is executed:

```
Long key =
    (Long)loadFromSession("CurrentArticleKey");
Papers.remove(key);
```

As you can see, the policy regarding rejected papers is harsh, leading to their immediate deletion. An interesting aspect is that a number of standardized methods are provided such as *loadFromSession*, which allows access to information stored by the previous state.

4.3 Security and Session Information

The second example used to illustrate the code generation mechanism is based on the classes representing security and session information. Similarly to the previous section, the technical metaelements are discussed first, followed by the business and model levels.

The metaclass *View* in Figure 5 has a metattribute *Access* that allows attributes of data type *String* and of multiplicity 0..n. However, unlike the metaattributes encountered so far, it turns into an attribute at the business level instead of at the actual model level. This means, that it will be an attribute slot to be filled with values at the actual model level.

[3] At the current state, the action language used is a subset of Java.

Therefore, in Figure 6, all views have an *Access* attribute, which is used to store, which roles are allowed to access that particular view. In Figure 10, *LibraryView* and *AdminView* make use of the *Access* attribute to provide access information. For example, *LibraryView* may only be accessed by those who have either the role *Subscriber* or the role *Admin*, because only these roles are specified as string values in the *Access* attribute.

The role of a user is stored in the session information. Instances of the metaelement *SessionInfo* represent an access point to all elements containing session data. For this purpose, at the technical level, a reference to zero or more *DataSets* is allowed. If any of these is an *SDataSet*, the *SecurityRoles* associated with it are available to the user whose *SessionInfo* contains the *SDataSet*.

In our example model (Figure 10), the only *SDataSet* associated with the *SessionInfo* instance is *User*. It contains two attributes that are instance of *Role*: *IsSubscriber* and *IsAdmin*. If the value of either of these is *true*, that user belongs to the respective group. As you can see, a subscriber would be able to access the *LibraryView* but not the *AdminView*.

5. CODE GENERATION PROCESS

So far we have described the basic principles of the OMEGA approach and the metamodel hierarchies used to illustrate it. In this section we will provide an overview of the architecture of the code generator.

The code generator backend is intended to perform a transformation of a set of static and dynamic OMEGA models into the resources constituting the modeled system, such as source code files and deployment descriptors. In comparison with open source tools for model-to-code transformation, such as OpenArchitectureWare[4] or AndroMDA[5], the OMEGA generator is able to leverage the advantages provided by the hierarchical nature of the underlying metamodels. For our implementation we have chosen a template-based model-to-code approach (cf. [13]). The generator uses source code templates that contain the target text and code to access the input models, therefore the templates closely resemble the target source code. Errors in the output of the generator can easily be traced back and be fixed at the template level. Additionally, due to the similarity between source code templates and the output in the target language, templates can be derived from a reference implementation, a prototypical example representing all relevant architectural aspects of a target system (cf. [13]).

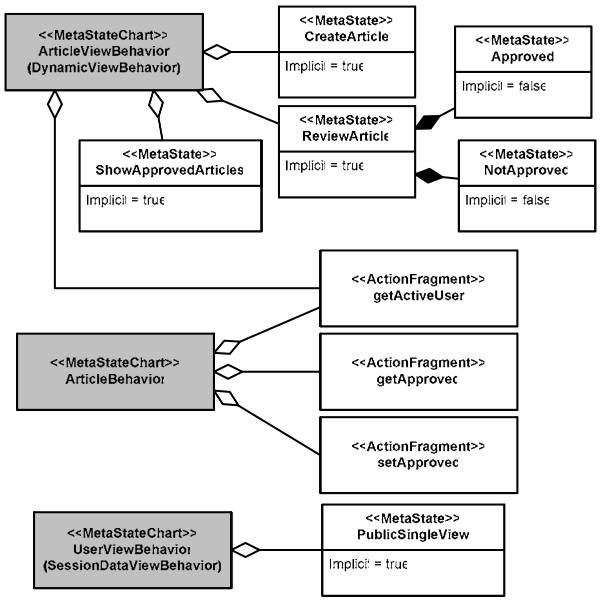


Figure 9 - The Dynamic Business Metamodel (Excerpt)

The template engine that has been used in the implementation, Apache Velocity, provides an untyped, interpreted language that contains basic control structures and the means to access Java objects but lacks advanced language features such as exception handling.

An overview of the steps of the code generation process is shown in Figure 3. In the OMEGA editor plugin, UML-compatible input models are designed by the modeler (see upper left corner). These class and statechart diagrams are passed to the code generation plugin.

As a first step, these models are transformed into the generator model, which has a format more suited to a template-based code generator (as shown in the upper right corner). Problematic aspects of the source model include the non-existence of optional

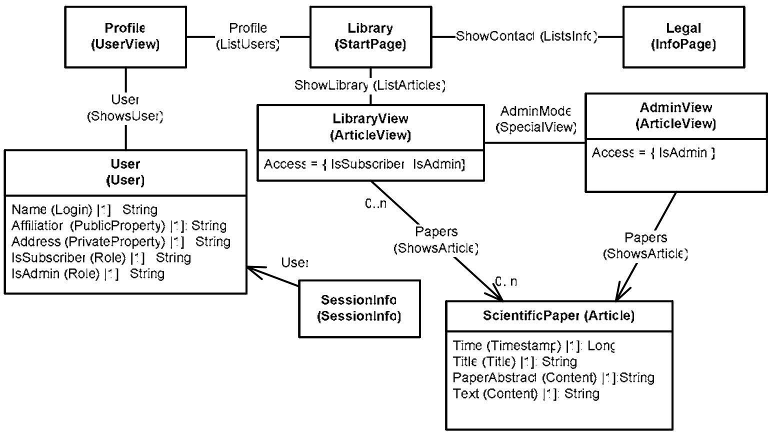

Profile
(UserView)
Profile
(ListUsers)
Library
(StartPage)
ShowContact (ListsInfo)
Legal
(InfoPage)
ShowLibrary (ListArticles)
User
(ShowsUser)
LibraryView
(ArticleView)
Access = { IsSubscriber IsAdmin }
AdminMode
(SpecialView)
AdminView
(ArticleView)
Access = { IsAdmin }
User
(User)
Name (Login) |*] String
Affiliation (PublicProperty) |*]: String
Address (PrivateProperty) |*] String
IsSubscriber (Role) |*] String
IsAdmin (Role) |*] String
0..n
Papers
(ShowsArticle)
Papers
(ShowsArticle)
User
SessionInfo
(SessionInfo)
0..n
ScientificPaper (Article)
Time (Timestamp) |*]: Long
Title (Title) |*]: String
PaperAbstract (Content) |*]:String
Text (Content) |*]: String

Figure 10 - Actual Static Model

[4] http://www.openarchitectureware.org

[5] http://www.andromda.org

associations, which may result in references containing a *null* value, and the fact that not all information is aggregated and organized in a way suitable for the generation process.

After the initial processing of the models has been performed, domain-specific transformations of the static and dynamic models are carried out, leading to a modified generator model M'. In the case of the static models this means that additional domain-specific attributes are woven into the model. The transformation is controlled via an XML definition of the attributes that are supposed to be added to classes of a particular metaclass. For example, all classes of type *Article* receive a new attribute called *Approved*, which is used for the review process.

Additionally, in this processing step patterns are resolved, i.e., the state-substate pattern in the dynamic model (see section 3.3). The transformations required for this step are also defined in XML, and offer options such as the introduction of new states to a dynamic model and the rerouting of transitions to incorporate these new states.

The final step in the generation process is the actual model-to-code transformation performed by the Velocity template engine (represented by the last arrow in Figure 3). For each node in the static models a set of templates is invoked according to an XML mapping associating metaclass names with output templates. A reference to this node is then passed to each of the invoked templates, which query the node object for its relevant properties such as name, attributes and associations with other nodes. For example, information about attributes provides the basis for attributes and/or getter and setter methods in a Java source file.

It is important to note that each input node from M' can result in any number of output resources, e.g., a node with the metaclass *DataSet* can be transformed into an EJB bean class and the various interfaces required. Conversely, a single output resource can also be generated from the extent of all classes instantiating the same metaclass, i.e., the XML deployment descriptor in an EJB environment.

Static nodes that are associated with dynamic models are subject to further processing. The generator applies the state pattern (cf. [14], [15]) to create the implementation of a state machine for every dynamic model.

All these artifacts are written to a prepared skeleton eclipse project that contains required libraries and a build script that creates a deployable, ready-to-run binary file of the modeled system.

While the process described above uses established code generation techniques, it allows for accessing the metainformation provided by the hierarchy. For example, domain-specific attributes and methods can be woven into classes based on their metaclasses. The separation of templates for the different layers (i.e., technical and business) has been implemented using basic model-to-model transformations and facilitates the rapid creation of additional business domain templates.

6. JAVA-SPECIFIC CODE GENERATION PROBLEMS

In the previous section we have provided an overview of the code generation process. In this section we will discuss some selected issues we have experienced during the implementation and evaluation of the code generation plugin.

For the purpose of evaluation we have created several examples targeted at J2EE/EJB2.x compliant application servers. This choice was made due to the multitude of services provided by a J2EE environment, such as persistence, transaction management, declarative, role-based security, etc. that were needed for the realization of the examples.

J2EE as a target platform has generally proven beneficial; however, some inherent characteristics of the platform complicated the development of the source code templates. The amount of classes and interfaces required to implement an EJB entity bean for example has been somewhat hindering during the initial creation of the source code templates, due to the fact that even simple changes required adapting various templates. We expect this situation to improve by adopting the Java Persistence API that simplifies the development of persistence-capable, transaction-safe classes.

Initial proof-of-concept implementations of our examples were closely tied to the EJB technology, because they were focused on demonstrating the applicability of our own Web framework COBANA rather than on efficient implementation. The basis for using alternative technologies, such as Hibernate, was created by refactoring our reference implementation, separating EJB-specific from other system aspects by introducing patterns such as Business Delegate or Data Transfer Object (cf. [16]).

In section 3.3 the notion of the explicit metastate was introduced as a state containing entry code supplied by the modeler for situations where none of the implicit metastates contained in the hierarchy is applicable. As the modeler is relatively free as to what kind of code to insert, the problem of gracefully handling potential exceptions arises. Due to the lack of a Java metamodel in the Velocity template engine, it is not possible to wrap critical parts of the code with the required specific exception handling code. The only way to ensure the correctness of the generated code is to wrap all modeler-defined entry code passages with generic exception handling. This situation was improved by the definition of technology-independent but meaningful exception types, i.e. for exceptions occurring during data access.

For the access control example described in section 4.3 the use of declarative servlet security mechanisms was planned. In order to be able to specify access restrictions for URL patterns, the passing of View names (cf. section 4.1) via query strings was replaced by a mechanism based on URL rewriting, a mechanism that enables a Web server to programmatically manipulate the URLs of incoming requests. In our case it allows for the representation of query strings in an URL-like fashion, e.g., *http://localhost/Controller?View=Library* can be expressed as *http://localhost/Controller/Library*. As URL patterns in the servlet specification are not supposed to be specified via regular expressions, all possible combinations of *Views* and *ChildViews* have to be declared in the security configuration.

On a more general level, the lack of an equivalent for UML-style bidirectional associations in the Java programming language requires representation as two unidirectional references causing some overhead and loss of semantics.

7. CONCLUSIONS

In this paper we have presented the OMEGA approach for the model-driven development of J2EE applications, which builds on the concepts promoted by Executable UML, i.e. the use of class and state chart diagrams for modeling executable systems. We have presented the experience we've gained from implementing a code generation facility using Java. Using examples from the domain of content management systems we have provided an overview on the code generation process. We have also presented an overview of some of the difficulties we have experienced in the prototypical implementation of the code generator and the solutions we have chosen to overcome these difficulties.

Our personal experience with the code generator has led to several interesting insights. First, it is very helpful to use a reference implementation as a basis for template design, which is subsequently parameterized to source code templates.

VTL has proven to be a good choice due to its simple syntax. While there are more powerful scripting languages, readability of the templates is absolutely essential and for this among other reasons, we have abandoned more complex approaches such as XSLT or code injectors such as InjectJ[8].

In the future we are planning to provide source code templates for additional target platforms, such as EJB3 or Hibernate and broaden the basis for a thorough evaluation of the approach by modeling and generating more examples. We are currently working on the introduction of a component specification facility that provides the means to specify and generate black box components.

8. ACKNOWLEDGMENTS

This work was supported by a grant from the Landesstiftung Baden-Wuerttemberg foundation to Michael Schwind.

9. REFERENCES

1. Haywood, D.: MDA: Nice Idea, Shame Abou the.: The ServerSide (2005)
2. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software **20** (2003) 19-25
3. Greenfield, J., Short, K., Cook, S., and Kent, S.: Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools (2004)
4. Booch, G., Brown, A., Iyengar, S., and Selic, B.: An MDA Manifesto. MDA Journal (2004)
5. Miller, J.a.M., J.: MDA Guide Version 1.0.1, OMG Document Number: omg/2003-06-01. (2003)
6. Mellor, S., Balcer, M.: Executable UML – A Foundation for Model-Driven Architecture. Addison-Wesley, Hoboken (2002)
7. Haywood, D.: MDA: Nice Idea, Shame about the…. The Server Side (2004)
8. Gitzel, R., Hildenbrandt, T.: A Taxonomy of Metamodel Hierachies - Working Paper 1-05. http://www.wifo.uni-mannheim.de/ ~gitzel/publications/taxonomy.pdf. (2005)
9. UML 2.0 Infrastructure Specification, ptc/03-09-15. (2003)
10. Gitzel, R.: Model-Driven Software Development Using a Metamodel-Based Extension Mechanism for UML, Vol. 28. Peter Lang Verlag, Frankfurt a.M. (2006)
11. Gitzel, R., Merz, M.: How a Relaxation of the Strictness Definition Can Benefit MDD Approaches With Meta Model Hierarchies. 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2004). SCI, Orlando, USA (2004)
12. Schwabe, D., Rossi, G.: An Object-Oriented Approach to Web-based Application Design. Theory and Practice of Object Systems **4** (1998)
13. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA (2003)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Niaz, I.A., Tanaka, J.: Code Generation From UML Statecharts. 7th IASTED International Conference on Software Engineering and Applications (SEA 2003), Marina Del Rey, USA (2003)
16. Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns. Best Practices and Design. Sun Microsystems, Inc., Mountain View, CA., USA (2003)

[8] http://injectj.fzi.de

M3PS: A Multi-Platform P2P System Based on JXTA and Java

Leonard Barolli
Department of Information and Communication Engineering
Fukuoka Institute of Technology (FIT)
3-30-1 Wajiro-Higashi, Higashi-Ku, Fukuoka 811-0295, Japan
Tel.: +81-92-606-4970
E-mail: barolli@fit.ac.jp

ABSTRACT
Peer-to-Peer computing offers many attractive features, such as collaboration, self-organization, load balancing, availability, fault tolerance and anonymity. In our previous work, we implemented a synchronous P2P collaboration platform called TOMSCOP. However, the TOMSCOP was implemented only in Windows XP OS. In this work, we extend our previous work and present a multi-platform Peer-to-Peer system. The proposed system operates very smoothly in UNIX Solaris 9 OS, LINUX Suse 9.1 OS, Mac OSX, and Windows XP. In this paper, we present the design of proposed system and four application tools: info, joint draw pad, shared web browser and subaru avatar.

Categories and Subject Descriptors
H.5.3 [**Information Interfaces and Presentation**]: Group and Organization of Interfaces ñ *collaborative computing, computer-supported cooperative work, synchronous interaction, web-based interaction.*

General Terms
Management, Design, Experimentation, Languages.

Keywords
JXTA, Java, Java Applications, P2P Systems, Multi-platform.

1. INTRODUCTION
Peer-to-Peer (P2P) computing offers many attractive features, such as collaboration, self-organization, load balancing, availability, fault tolerance and anonymity. Collaborative computing, usually known as groupware or Computer Supported Collaborative Work (CSCW), refers to technologies and systems that support a group of people engaged in a common task or goal and provide an interface to a shared environment. Grudin [1] defined a time/location matrix to categorize collaborative systems as four types, among which one called distributed synchronous collaborative system can support a group of people in different locations to conduct a common task or activity at the same time.

The fundamental element in a synchronous collaborative system is the shared application, where multi-users can synchronously view and manipulate with the mode of what you see is what I see (WYSIWIS) [2].

The shared applications fall into two categories, screen-copy system and event-aware system [3]. The former allows many existing single-user applications to be used by multi-users in cooperative fashion via capturing an application window and sending it as image data similar as a video camera. Typical collaboration-transparent systems are Sun ShowMe, Microsoft NetMeeting, and Intel ProShare.

There are generally three types of connection and message passing topologies between multiple usersí computers/devices used for their collaborations. One is called a centralized topology in which there is no direct connection between computers and all messages are mediated by an inter-mediator generally known as a group server. VCR [4], Habanero [5], Worlds [6], TANGO [7], TeamWave [8] adopted this topology. Such connection topology follows the ordinary client/server model. In these systems, the server is the system bottleneck, thus the whole system may be down when the server has some troubles. The P2P model is used for hybrid topology and decentralized topology [9,10]. In hybrid topology a peer needs to connect to both a group server and other peers. In decentralized topology every peer is able to directly connect to all other peers and messages are sent without intermediation of a server. Groove [11] and Endeavorsí Magi [12] have adopted the hybrid topology. Even they overcome some drawbacks of client/server-based systems, a peer has yet to go to the server and strictly follows the procedures defined by a particular system.

In our previous works [13, 14, 15], we proposed and built some P2P collaborative systems. TOMSCOP system [13] is based on JXTA framework that consists of the virtual JXTA network and basic peer group services [16, 17]. JXTA is a general framework able to support a broad range of P2P computing such as distributed computation, storage, agent, content distribution, and system test. On the JXTA open source web site [18] there are many JXTA-based projects. On the top of JXTA framework, TOMSCOP provides four types of services: synchronous message transportation, peer room administration, peer communication support and application space management for development of shared applications and creation of collaborative communities. However, the TOMSCOP was implemented only in Windows XP.

In this paper, we extend our previous works and present the implementation of a Multi-Platform P2P System (M3PS). In order that M3PS operates in multiplatform, we designed and implemented three new functions: look and feel, mouse button and room information. The proposed system operates very smoothly in UNIX Solaris 9 OS, LINUX Suse 9.1 OS, Mac OSX, and Windows XP OS. We present four application tools: INFO, Join Draw Pad (JDP), Shared Web Browser (SWB) and Subaru Avatar (SA). The M3PS system can be used also for other applications [19].

This paper is organized as follows. In Section 2, we give a brief description of TOMSCOP. In Section 3, we introduce our proposed M3PS system. In Section 4, we present some application tools of M3PS. Finally, we conclude the paper in Section 5.

2. TOMSCOP OVERVIEW

The TOMSCOP is a event-aware system but with a complete different connection topology and other special features. To provide necessary services, the TOMSCOP platform is developed as a bridge between JXTA framework, shared applications and collaborative communities as shown in Fig. 1.

TOMSCOP is designed using the metaphor of center-room-facility. Users or peers gather in a virtual community center to meet each other, enter some rooms corresponding to specific groups of interests, and work together using available facilities, i.e., shared applications. As shown in Fig. 2, the platform provides four kinds of services:

- **Synchronous Message Transportation** to transport all messages between peers in the center and inside rooms based on the JXTA pipe service.
- **PeerRoom Administration** to administrate peers and rooms in the center, and promptly shows the core awareness information of peers and rooms via using presence control and identity control.
- **PeerCommunication Support** to provide a set of communication channels for different media. The JMF (Java Media Framework) technology is used for audio and video communications.
- **ApplicationSpace Management** to manage usages of virtual center space and common shared applications in order to maintain good harmonization in collaboration among multi-users in a virtual room but physically in different places.

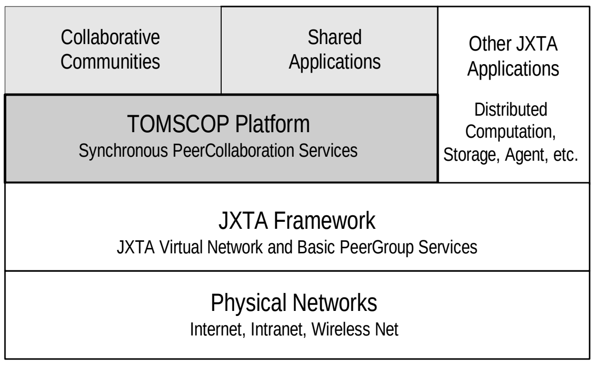

Collaborative Communities
Shared Applications
Other JXTA Applications
TOMSCOP Platform
Synchronous PeerCollaboration Services
Distributed Computation, Storage, Agent, etc.
JXTA Framework
JXTA Virtual Network and Basic PeerGroup Services
Physical Networks
Internet, Intranet, Wireless Net

Figure 1. TOMSCOP platform.

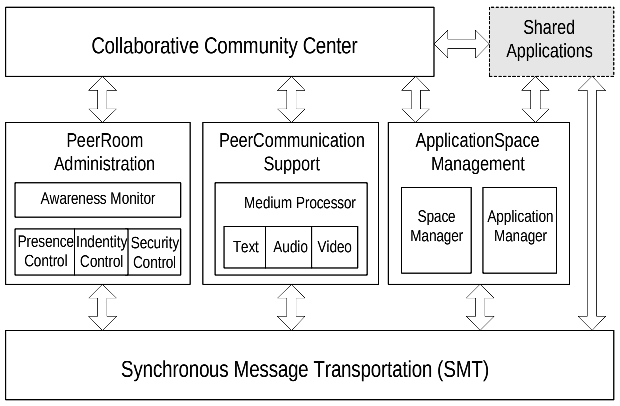


Figure 2. TOMSCOP architecture.

3. PROPOSED SYSTEM

We have implemented M3PS in our lab in four OS. The M3PS environment includes two Workstations Sun Blade 1500 (OS: Solaris 9; CPU: 1.062 GHz UltraSPARC IIIi, HD: 80GB, Memory: 512 MB), three note book PCs (OS: Windows XP; CPU: Pentium M 1.5GHz, HD: 40GB; Memory: 768 MB), one note book PowerBook G4 PC (OS: MacOSX Ver.10.3.4; CPU: PowerPC G4 867 MHz; HD: 40GB; Memory: 256 MB) and two Desktop computers (OS: SUSE Linux 9.1; CPU: Pentium 4 2.60GHz; HD: 80GB; Memory: 1GB).

For the system implementation, we used Java language. The P2P was implemented based on JXTA framework. We changed the following three functions (shown in Fig. 3) in order to make the system operates in multiplatform: Look and Feel function; Mouse Button function; and Room Information function.

- **Look and Feel Function**

In TOMSCOP, this function was implemented only for Windows XP OS, we changed it to adapt for different systems in dynamic way.

- **Mouse Button Function**

In TOMSCOP, this function was implemented for double button, because it was for Windows XP OS. In M3PS, we changed in single button for Mac OS and triple button for UNIX and Linux. In order to operate in multiplatform the mouse button function should have single and multiple modes.

- **Room Information Function**

In TOMSCOP, the room information and community were the same. In M3PS, we separated the room information and community.

Functions Systems	TOMSCOP	M3PS
Look & Feel	Windows XP	Multiplatform
Mouse Button	> Single	≥Single
Room and Community	Single Frame	Different Frames

Figure 3. M3PS improved functions.

In following, we will explain in detail the architecture of the proposed M3PS. The M3PS provides four services: Synchronous Message Transportation (SMT); PeerRoom Administration (PRA); PeerCommunication Support (PCS) and ApplicationSpace Management (ASM).

3.1 Synchronous Message Transportation

The SMT of M3PS consists of two modules: a message sender and a message receiver, as shown in Fig. 4. The message sender includes three main functional components: a data collector, a message encoder and a message pusher. The message receiver includes three correspondent components, a data distributor, a message decoder and a message listener. The pipes are abstractions of JXTA data transmission route on the JXTA virtual network. There are three basic pipes: insecure unicast, secure unicast and propagate types. The propagate pipe is used in SMT for multicasting. In addition to the pipes used for group communication inside rooms, a peer is able to exchange messages with any individual peer and all peers in a collaborative community, which is a collection of peers and rooms. Therefore SMT provides three categorized transmission modes:

- Room mode: for group message multicasting among shared applications in a room;
- Community mode: for message broadcasting to all peers in a community;
- One-to-One mode: for one-to-one private message exchange between any two peers in a community.

Thus, we have built three kinds of propagate pipes corresponding to these modes. When receiving attribute data and/or primitive data from the service modules or shared applications, the messages will be formed based on the format shown in Fig. 5. The Coding ID (CID) is used to extract the primitive data as well as other data from a message. The Application ID (AID) is used for uniquely identifying an application for correct data dispatch. The Object ID (OID) and the source peer name elements are optional for being used by shared application developers according to their actual requirements.

The following four methods to send primitive data to all peers in a room have been provided.

Type A: **sendMsg**(dest., source, CID, AID, OID)
ñ to send attribute data related to an application

Type B: **sendMsg**(dest., source, CID, AID, OID, string[])
ñ to send a string array used for chat text

Type C: **sendMsg**(dest., source, CID, AID, OID, integer[])
ñ to send integer array like mouse coordinates

Type D: **sendMsg**(dest., source, CID, AID, OID, byte[])
ñ to send binary file, audio and video data

3.2 PeerRoom Administration

Due to no server in our system, each peer is able to administrate groups. This is a distinctive characteristic of M3PS compared with others centralized and hybrid collaborative systems. In our system, such work is done by the services of PRA, which is responsible for room creation, publication and searching under supported by the JXTA. Any room created by a peer must have a unique name and ID number, and its associated group advertisement should be generated and published on the JXTA network so that other peers can find the room by searching the room advertisement.

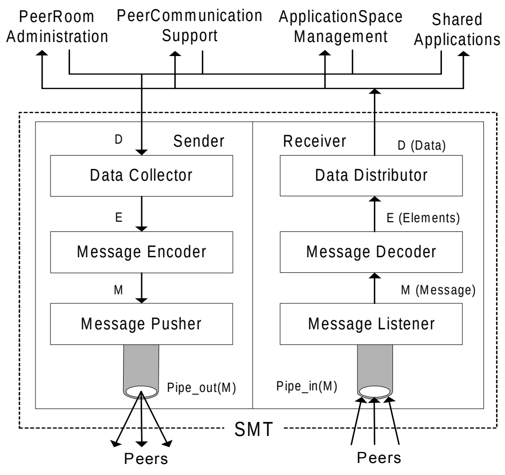


Figure 4. Message sender and receiver.

Field Name	Value		
Transmission Mode	Room	Community	One-to-One
Destination	Room/Peer	" Community "	Peer
Source	Peer	Peer	Peer
Coding ID (CID)	Integer	——	——
Application ID (AID)	Integer	Integer	Integer
Object ID (OID)	Integer	Integer	Integer
Element	String, integer, byte	String, integer, byte	String, integer, byte

Figure 5. Message formats.

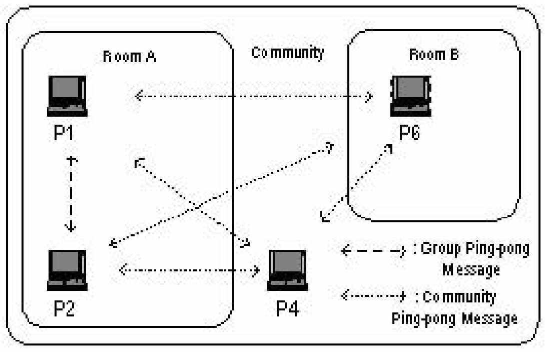


Figure 6. Presence control.

It is very important and necessary to provide prompt and correction awareness information of peers and rooms as well as their dynamic changes. This function is provided by the awareness monitor, which automatically and periodically collects status information of peers and rooms in the community using presence control and identity control. The presence control shows the peerís presence information. As mentioned above, it is difficult to administrate peersí presence in real-time only using advertisements provided by JXTA core service. To provide correct presence information, the ping-pong detection approach is used to manage peersí presence in real-time as shown in Fig. 6.

Peersí normal login and logout messages are sent using the method of Type-A sendMsg( ). In the same way a community ping-pong message is sent using community message mode. A peer who logins to or logouts from the JXTA network sends a message to all other peers to inform its existence or absence. In addition, the peer sends a ping message to all other peers in the same room and community every 2 minutes. If a peer does not reply for several requests, it will be automatically excluded from the community/room.

A peer in a room can be assigned some roles such as a group leader or member for synchronous collaborative work. Peersí roles are managed by the identity control as follows.

- Chair to control the identities of other peers and also play a coordinative role in a room.
- Player to be able to control shared space and manipulate shared applications, such as a game player or meeting presenter in the real world.
- Observer to only watch the shared space and applications but have no right to manipulate the space and applications.

3.3 PeerCommunication Support

The PCS provides built-in communication tools to collaborative processes for administrating rooms and using shared applications. The SMT offers three transmission service modes: community, room and one-to-one to send a message to all peers, room peers, and an individual peer, respectively. Each mode uses its own pipe corresponding to a related pipe advertisement.

3.4 ApplicationSpace Management

ASM consists of a space manager to control the shared space, and an application manager to control operations on shared applications. Both managers are related to peersí identities (chair, player and observer) in a room as shown in Fig. 7. A presenter peer in a room has initiative to operate a virtual space or shared applications, but others can only see what he/she has done. A chair is able to change other peersí identities.

4. M3PS APPLICATION TOOLS

In this section, we present four application tools of M3PS: INFO, JDP, SWB and SA.

4.1 INFO

The INFO application tool gives the information of the peers in M3PS community. So, if a peer is logged in the system, the other peers can get the information and communicate together. In the top of main window of INFO tool are shown also other tools such as JDP, SWB and SA. Therefore, if a peer wants to shift to other applications, he can use this panel to change his mode to other tools. In the center of INFO window are shown: Room Name, Room Description, Room ID, Chair Peer and Presentation Peer. On the top in left side frame are shown: Peer Name, Room Name, Collaboration Mode and Peer Identity. In the middle frame (left side) is shown Conference Room and in the bottom frame Community information. The display captures of INFO for Linux, and UNIX are shown in Fig.8 and Fig. 9, respectively.

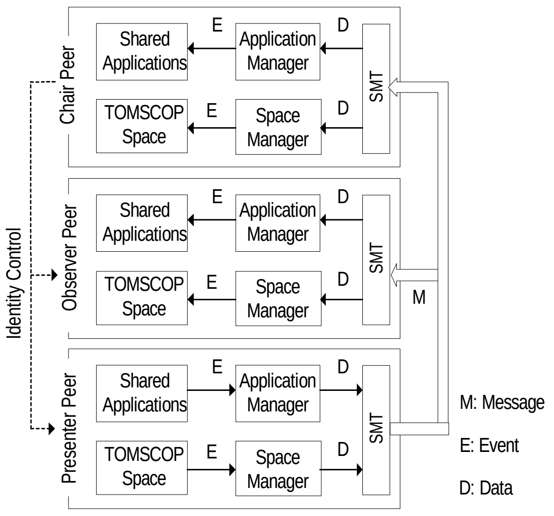
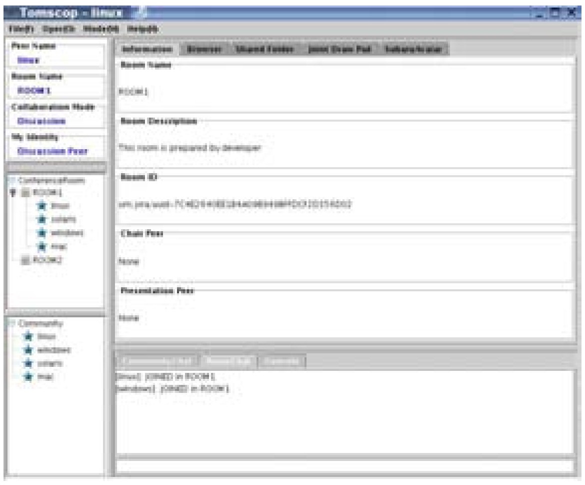


Figure 7. ApplicationSpace management.



Tomscop - linux

File(f) Open(O) Mode(M) Help(H)

Peer Name: linux
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

Information | Browser | Shared Folder | Joint Draw Pad | SakuraAvatar

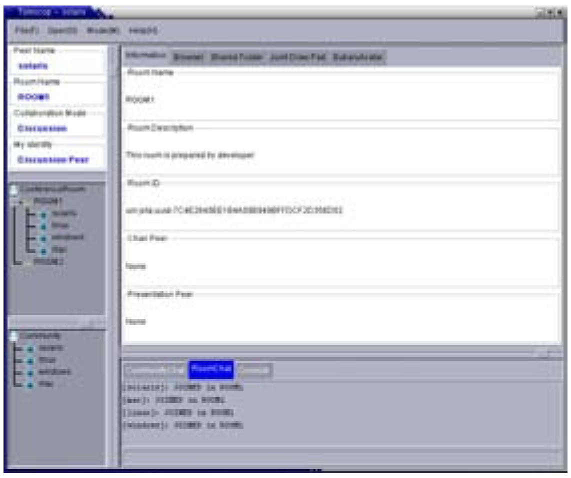
Room Name: ROOM1
Room Description: This room is prepared by developer
Room ID: urn:jxta:uuid-7C4E2940EE1B4A0989498FDCF2D35ED02
Chair Peer: None
Presentation Peer: None

ConferenceRoom
ROOM1
linux
solaris
windows
mac
ROOM2

Community
linux
windows
solaris
mac

[linux]: JOINED in ROOM1
[windows]: JOINED in ROOM1

Figure 8. INFO in Linux.



Tomscop - solaris

File(f) Open(O) Mode(M) Help(H)

Peer Name: solaris
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

Information | Browser | Shared Folder | Joint Draw Pad | SakuraAvatar

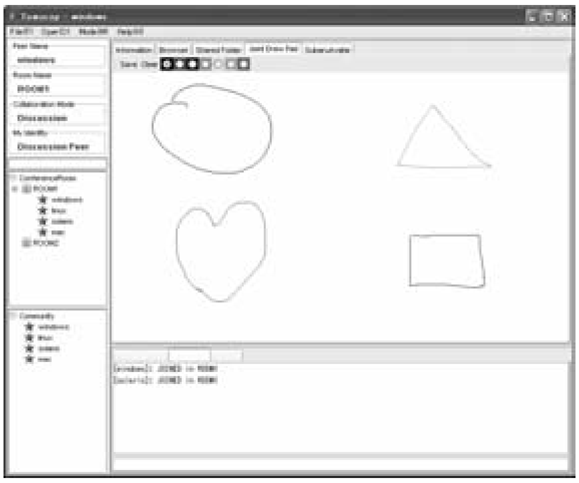
Room Name: ROOM1
Room Description: This room is prepared by developer
Room ID: urn:jxta:uuid-7C4E2940EE1B4A0989498FDCF2D35ED02
Chair Peer: None
Presentation Peer: None

ConferenceRoom
ROOM1
solaris
linux
windows
mac
ROOM2

Community
solaris
linux
windows
mac

[solaris]: JOINED in ROOM1
[mac]: JOINED in ROOM1
[linux]: JOINED in ROOM1
[windows]: JOINED in ROOM1

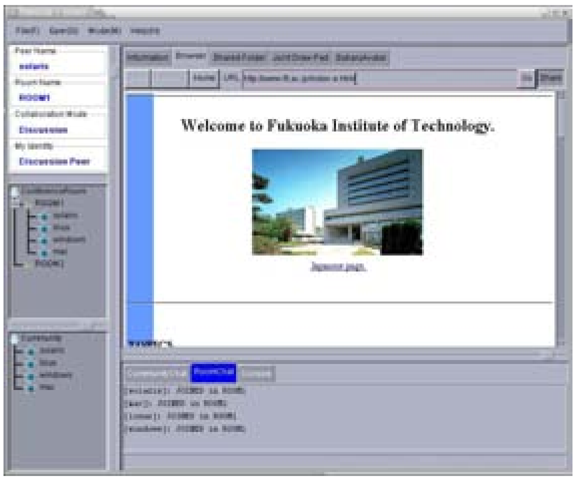
Figure 9. INFO in UNIX.


Tomscop - windows
File(F) Open(O) Mode(M) Help(H)
Information Browser Shared Folder Joint Draw Pad SubaruAvatar
Save Clear
ConferenceRoom
ROOM1
windows
linux
solaris
mac
ROOM2
Community

Peer Name: windows
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

```
[windows]: JOINED in ROOM1
[solaris]: JOINED in ROOM1
```

Figure 10. JDP in Windows XP.



Peer Name: solaris
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

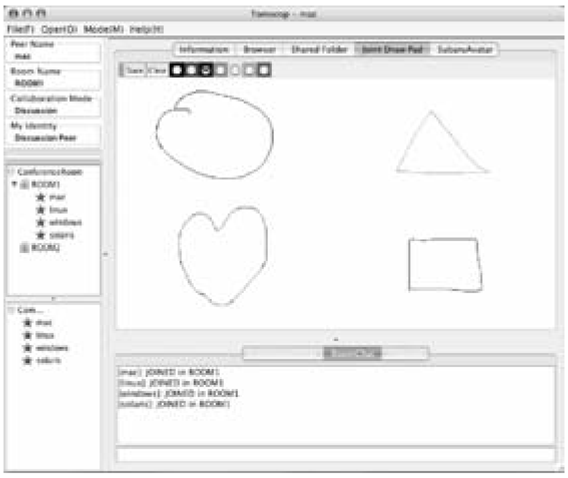
Welcome to Fukuoka Institute of Technology.

Japanese page.

TOPICS

```
[solaris]: JOINED in ROOM1
[mac]: JOINED in ROOM1
[linux]: JOINED in ROOM1
[windows]: JOINED in ROOM1
```

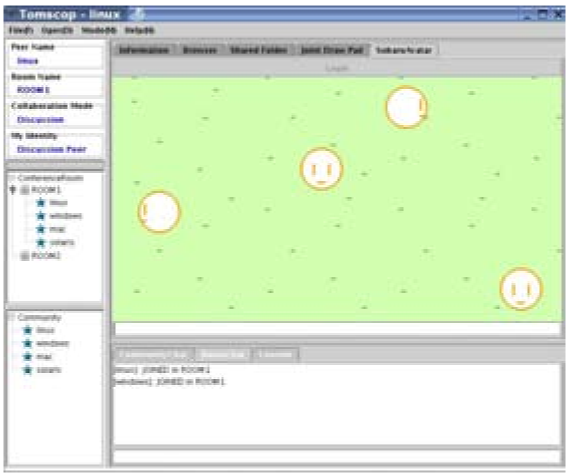
Figure 13. SWB in UNIX.


Tomscop - mac
File(F) Open(O) Mode(M) Help(H)
Information Browser Shared Folder Joint Draw Pad SubaruAvatar
Save Clear
ConferenceRoom
ROOM1
mac
linux
windows
solaris
ROOM2
Com...

Peer Name: mac
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

```
[mac]: JOINED in ROOM1
[linux]: JOINED in ROOM1
[windows]: JOINED in ROOM1
[solaris]: JOINED in ROOM1
```

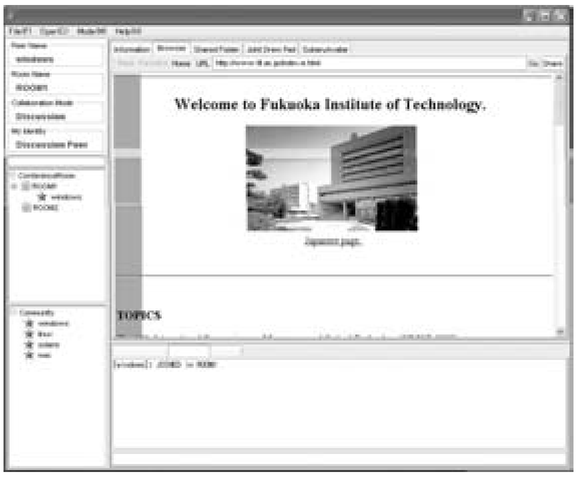
Figure 11. JDP in Mac.


Tomscop - linux
File(F) Open(O) Mode(M) Help(H)
Information Browser Shared Folder Joint Draw Pad SubaruAvatar
Login
ConferenceRoom
ROOM1
linux
windows
mac
solaris
ROOM2
Community

Peer Name: linux
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

```
[mac]: JOINED in ROOM1
[windows]: JOINED in ROOM1
```

Figure 14. SA in Linux.


File(F) Open(O) Mode(M) Help(H)
Information Browser Shared Folder Joint Draw Pad SubaruAvatar
Home URL
Go Share
ConferenceRoom
ROOM1
windows
ROOM2
Community

Peer Name: windows
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

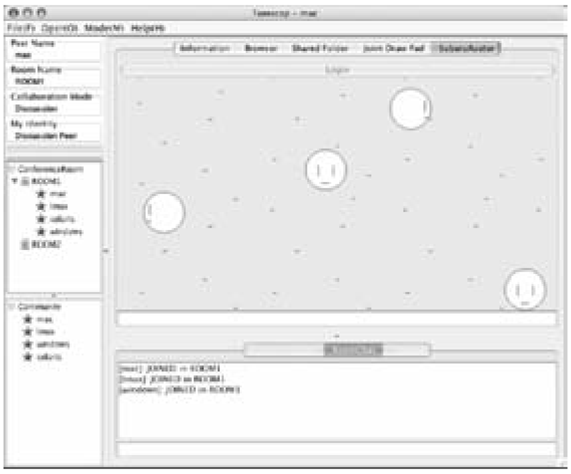
Welcome to Fukuoka Institute of Technology.

Japanese page.

TOPICS

```
[windows]: JOINED in ROOM
```

Figure 12. SWB in Windows XP.


Tomscop - mac
File(F) Open(O) Mode(M) Help(H)
Information Browser Shared Folder Joint Draw Pad SubaruAvatar
Login
ConferenceRoom
ROOM1
mac
linux
solaris
windows
ROOM2
Community

Peer Name: mac
Room Name: ROOM1
Collaboration Mode: Discussion
My Identity: Discussion Peer

```
[mac]: JOINED in ROOM1
[linux]: JOINED in ROOM1
[windows]: JOINED in ROOM1
```

Figure 15. SA in Mac.

4.2 Joint Draw Pad

The JDP is a tool for making joint figures or designs. The users may be in different locations but they can draw or make a design in the same pad. For the sake of space, we show in Fig. 10 and Fig. 11 the display for Windows XP and Mac. In fact, in Fig. 10 (Windows XP) is drawn the circle and this circle is shown at the same time in other environments. The heart is drawn in Linux, the rectangular shape is drawn in Mac, and the triangular shape is drawn in UNIX. This tool also can be used for collaboration research from different sites.

4.3 Shared Web Browser

The SWB application tool is shown in Fig. 12 and Fig. 13 for Windows XP and UNIX, respectively. Using this application, the peers in the same room can see the same URL. In these figures is shown the homepage of Fukuoka Institute of Technology (FIT), Japan. This tool is very useful application and can be used for collaboration research. For example if one peer finds an interesting homepage with good papers and research work, he can send this information to other peers. Thus, they can work efficiently together.

4.4 Subaru Avatar

Another application tool is SA. For using this tool, the peer prepares a room, then changes from Mode to Discussion. Next, he pushes the Login button. Then, when the user moves the mouse in the direction he wants, the avatar follows the mouse movement. In this application, there is a text field. If the peer writes something in the text field, the information will be shown in the peerís avatar. In this way, the peers can communicate and exchange the information with each other. The capture display for Linux and Mac are shown in Fig. 14 and Fig. 15, respectively. The SA can be used as a chat application between peers for communication in real time.

5. CONCLUSIONS

In this paper, we improved our previous platform TOMSCOP and implemented a multiplatform P2P system called M3PS. Different from many other similar platforms, it has adopted a pure P2P architecture and each peer has to administrate collaborative rooms. To demonstrate the platform effectiveness, we showed four application tools: INFO, JDP, SWB and SA. By many experiments, we found that proposed M3PS operates smoothly in the four environments.

As the future work, we will deal with following issues:

- secure room administration in P2P communication;
- audio and video communications;
- efficient pipe advertisement creation to avoid creating duplicated pipe advertisement for a room;
- development of new shared applications.

6. REFERENCES

[1] Grudin, B. *Computer-Supported Cooperative Work*, IEEE Computer, 1994.

[2] Steinmetz, R., and Nahrsted, K. *Multimedia: Computing, Communications and Applications*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 1995.

[3] Begole, J.B. Usability Problems and Causes in Conventional Application-Sharing Systems. Available on line at http:// simon.cs.vt.edu/begolej/Papers/, 1999.

[4] Ma, J., Huang, R., and Nakatani, R. Towards a natural internet-based collaborative environment with support of object physical and social characteristics. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, 11, 2 (2001), 37-53.

[5] Chabert, A., et. al. Java object-sharing in Habanero. *Communications of ACM*, 41, 6 (1998), 69-76.

[6] Mansfield, T., et al. Evolving orbit: a progress report on building locales. In *Proc. of the Group'97 International Conference*. ACM Press, 1997, 241-250.

[7] Beca, L., et al. TANGO - a collaborative environment for the World-Wide Web. Available on line at http://trurl.npac. syr.edu/tango/.

[8] Greenberg, S., and Roseman, M. *Using a room metaphor to ease transitions in groupware*. Technical Report 98/611/02, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, January 1998.

[9] Barkai, D. Peer-to-peer computing: technologies for sharing and collaborating on the net. Intel Press, 2001.

[10] Leuf, B. *Peer-to-Peer: Collaboration and Sharing over the Internet*. Addison-Wesley, 2002.

[11] Edwards, J. *Peer-to-Peer Programming on Groove*. Addison-Wesley, 2002.

[12] Magi, Available on line at http://www.endeavors.com/.

[13] Kawashima, T., Ma, J. TOMSCOP-A synchronous P2P collaboration platform over JXTA. In *Proc. of IEEE ICDCS-2004/MNSA-2004*, 2004, 85-90.

[14] Ma, J., Barolli, L., Shizuka, M., and Huang, R. A pure P2P synchronous collaborative system. *Journal of Applied System Studies (JASS)*, 5, 2 (July 2004), 133-145.

[15] Takata, K., Ma, J. GRAM - a P2P system of group revision assistance management. In *Proc of IEEE AINA-2004*, Fukuoka, Japan, March 2004, 587-592.

[16] Project JXTA, Available on line at http://www.jxta.org/.

[17] Gradecki, J.D. *Mastering JXTA: Building Java Peer-to-Peer Applications*. Wiley Pub., 2002.

[18] Traversat, B., et al. Project JXTA 2.0 Super-Peer Virtual Network, Sun Microsystems, Inc, 2003. Available on line at http://www. jxta.org/servlets/DomainProjects.

[19] Ma, J., Yang, L.T., Apduhan, B.O., Huang, R., Barolli, L., Takizawa, M. Towards a smart world and ubiquitous intelligence: a walkthrough from smart things to smart hyperspaces and UbicKids. *Journal of Pervasive Computing and Communications*, 1, 1 (March 2005), 53-68.

Mapping Clouds of SOA- and Business-related Events for an Enterprise Cockpit in a Java-based Environment

Daniel Jobst
TietoEnator Deutschland GmbH
Digital Innovations
80335 München, Germany
Daniel.Jobst@tietoenator.com

Gerald Preissler
Deutsche Post AG
SOPSOLUTIONS®
53250 Bonn, Germany
G.Preissler@deutschepost.de

ABSTRACT

This paper is about business process management (BPM) and business activity monitoring (BAM) using event processing. We will show why the management of business processes is important for all further steps towards an event-driven and real time enterprise. That is process automation using workflow engines and standards like the Web Service Business Process Execution Language (BPEL). As an underlying middleware platform we use the service oriented platform SOPware of Deutsche Post AG.

Events are emitted from all layers, the middleware platform layer and the business process layer, and figuratively build “event clouds”. Event processing functionalities will correlate both “event clouds” and feed business activity monitoring. There, enterprise performance cockpits and dashboards depict the performance of the enterprise and of its processes.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: patterns – *service oriented architecture, event processing.* D.2.8 [**Metrics**]: Process metrics – *business activity monitoring, enterprise cockpit.* D.3.2 [**Language Classifications**]: Object-oriented languages – *Java, J2EE.*

General Terms

Management, Measurement, Performance, Design, Standardization.

Keywords

Service oriented architecture, event processing, business activity monitoring.

1. TECHNOLOGIES AND ARCHITECTURE

In the following chapters we will concentrate on how to do monitoring better rather than the question what it is for. We will take into account recent developments in IT architecture and event processing technology.

1.1 Business process management (BPM)

According to the latest Gartner research paper on process management, today BPM is more than a collection of software tools. BPM is a management discipline with modeled business processes as its fulcrum. It aims to improve enterprise agility and the operational performance. Decisive BPM technologies are process modeling, process execution, a BPM suite, and accessing underlying resources using service orientated technologies. [1]

As modeling tool and modeling environment we are using the ARIS toolset in our use case. Processes in their finest granularity will be denoted in the Event-driven Process Chain (EPC) .

According to [2] and the monitoring and event processing approach we are discussing in this paper, focusing on processes and explicit process models are vital for further process measurement and monitoring.

1.2 Process automation and service orientation

According to the approach we are introducing in this paper, process automation needs some preconditions.

First of all, it needs process models in the right granularity for SOA. The modeled process steps have to match the appropriate service calls or sub processes in the later automated processes.

Then it needs a service oriented platform with concepts providing for services like authorization, authentication, user management, data management, logging, exception handling, monitoring, and security. SOPware is a Java-based infrastructure that allows the easy integration of J2EE and J2SE-based applications in an SOA environment.

Thirdly, process automation needs a standardized process language which can be executed by a workflow engine much like a programming language. The Business Process Execution Language is such a language which is designed to orchestrate web services. For the current version 1.1 (BPEL4WS) and version 2.0 (WSBPEL) see [4].

1.3 Business activity monitoring (BAM)

BAM as a technology gives access to key business metrics. It is used to monitor business objectives, to evaluate operational risk, and to reduce the action time between an event happening and the actions taken [5].

According to [6], BAM incorporates the different technologies like BPM, BPEL, event processing, and SOA. It links events, services, and processes with rules, notifications, and human interaction.

Results of BAM are needed for controlling an enterprise and are important for other methodologies like the balanced scorecard (BSC) and Six Sigma or for compliance with regulations like the Sarbanes Oxley Act (SOX) or Basel II.

1.4 Event processing (EP)

Each enterprise has to deal with a huge number of events. These events can be low level network events or high level business events. All of them have some kind of representations in the IT system. In order to deal in real time with a huge number of events, EP provides different technologies and methodologies. One is to classify events in different hierarchical layers, to look for patterns of events within a layer, and propagate new complex events to higher layers as introduced by [7]. Another way is to arrange events into streams and monitor, analyze, and act upon events as they appear in the stream [8].

EP is not a new technology. But with increasing hardware and network capacities and evolving standards and tools it gets a new momentum [9].

2. USE CASE

An abridged business process from the banking domain will be the consistent process example in this paper. It is a simple version of a credit application process. Once a credit application is received, a data validation service will be called. Then, depending on the amount of the application, either the scoring and approval will be done automatically or manually by an employee. After that the customer will be sent an email.

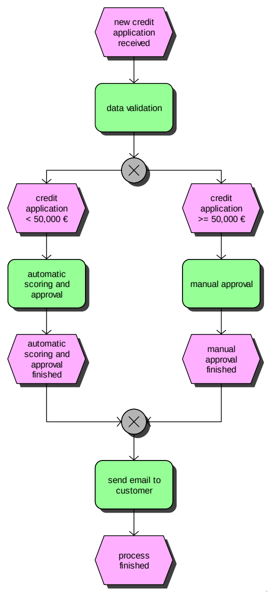

new credit application received
data validation
credit application < 50,000 €
credit application >= 50,000 €
automatic scoring and approval
manual approval
automatic scoring and approval finished
manual approval finished
send email to customer
process finished

Figure 1. Process in EPC notation

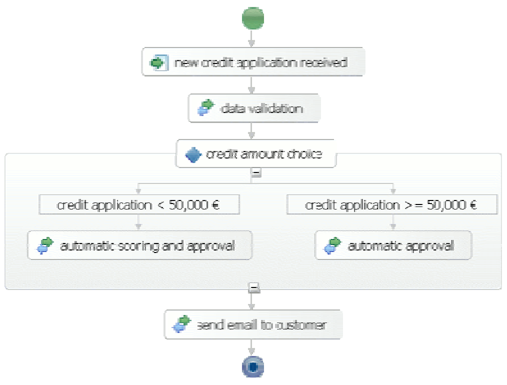

new credit application received
data validation
credit amount choice
credit application < 50,000 €
credit application >= 50,000 €
automatic scoring and approval
automatic approval
send email to customer

Figure 2. BPEL Process

Figure 1 shows the process in the EPC notation, Figure 2 is the process translated into BPEL. For our services "data validation", "automatic scoring and approval", "automatic approval", and "send email to customer" in the example BPEL process in Figure 2 there is a service implementation within SOPware which will be called by the BPEL workflow engine.

The BPEL process will be published as a service itself. In terms of multi channeling this is important because independent of a channel (e.g. portal, call center, self service devices) always the same process is used. And it is important for the monitoring. The different channels will be a dimension which will be monitored and analyzed in the BAM environment.

The basic concepts of a service-oriented architecture as well as the SOA implementation chosen for our studies are described in chapter 3.

The overall architecture of the approach we introduce in this paper is shown in Figure 3.

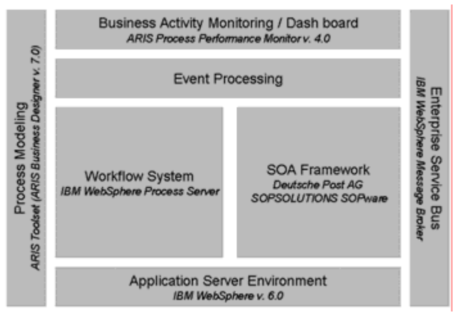

Business Activity Monitoring / Dash board
ARIS Process Performance Monitor v. 4.0
Event Processing
Workflow System
IBM WebSphere Process Server
SOA Framework
Deutsche Post AG
SOPSOLUTIONS SOPware
Application Server Environment
IBM WebSphere v. 6.0
Process Modeling
ARIS Toolset (ARIS Business Designer v. 7.0)
Enterprise Service Bus
IBM WebSphere Message Broker

Figure 3. Our architecture

The workflow system, the SOA framework, and the enterprise service bus are based on a J2EE application server environment. The workflow system runs the BPEL processes and publishes business and process events. The SOA framework is Java based too and publishes the lower level SOA events. The enterprise service bus (ESB) is responsible for event transportation and format transformations through the different layers.

Here we have to understand the problem: BAM tools monitor and analyze events from an high level "event cloud" whereas technical monitoring processes events form an "event cloud" of lower level events. Figure 4 shows the two different monitoring views with their layers and typical vendors and tools. With EP both views can be mapped in order to reach a consistency in terms of monitoring completely end-to-end without replacing existing installations by mapping instances.

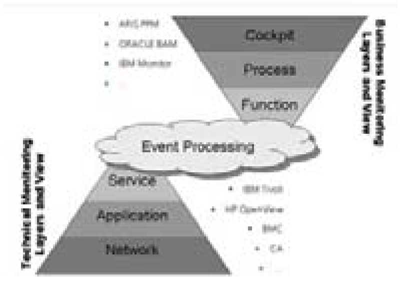


Figure 4. Technical and business monitoring layers and views

The event processing layer will correlate the actual business and process events (from the business instance) with the events from the technical SOA events (from the implementation layer). This we will describe in chapter 5.

As BAM tool we use the ARIS Process Performance Manager (PPM) to build personalized views of a performance dashboard to visualize key performance indicators (KPI) such as "average process runtime". The event processing layer will deliver the events and metrics needed for the dashboard and to be able to drill down to aggregated process views or to single process instances.

3. SOP SERVICE ORIENTED ARCHITECTURE MODEL

To discuss the events generated in a service oriented architecture (SOA), an understanding of the components of a SOA and their interactions is necessary. As of the time of writing, there are no standard or established definitions for this. The following paragraphs will try to define the characteristics of a SOA that are relevant for the topic of this paper. This is not intended to be a discussion of SOAs in general – which would be beyond the scope of the document – but to foster a common understanding of the problem domain under discussion.

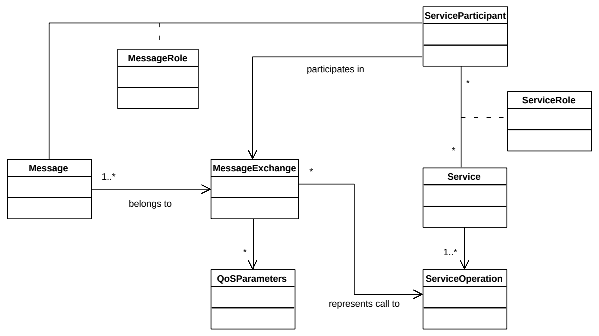


Figure 5. Elements of a SOA

Figure 5 shows an overview of the entities that are relevant for management purposes.

A service is the encapsulation of a defined unit of business logic. From a management point of view, a service is a set of service operations. Further functional properties such as the exact syntax of individual operations or their semantics are not considered here.

An entity acting within a SOA is called service participant. Each service participant can act as service provider or service consumer for one or more services.

The basic mechanism for service usage is the exchange of XML-encoded service messages between a service consumer and a service provider.

A message exchange groups one or more transmissions of individual messages that together complete one invocation of a service operation. The pattern of messages exchanged between consumer and provider during an exchange is a property of the operation. A number of such exchange patterns are defined in [17].

For each message exchange, a number of quality of service properties (QoS) can be defined. A common example for a QoS property is the maximum response time for an In-Out type operation. These properties can be defined in multiple ways, e.g. for all calls to a service operation or by individual negotiation between service provider and consumer. At runtime, a concrete set of QoS properties applies to an individual message exchange. The adherence to the parameters defined for each exchange should be monitored.

For studying the actual integration of a SOA implementation into a business activity monitoring solution, the service-oriented platform SOPware, provided by the `SOPSOLUTIONS` department of Deutsche Post AG, has been chosen. This platform provides a distributed component called SBB library that each service participant uses to consume or provide services. This component provides management and monitoring capabilities based on the Java Management Extensions. Part of these facilities is the generation of SOA-releted events as described below. In addition to the SOPware internal processing and reporting capabilities, this information can be propagated to existing monitoring or event-processing environments.

This paper focuses on a small detail of SOPware, for further information about the platform as a whole, please see [18].

4. EVENTS IN A SERVICE ORIENTED AND PROCESS ENVIRONMENT

4.1 Events

"An event is an object that is a record of an activity in a system."[7] This can be both a more technical event like "service response is five seconds overdue" and an event of a higher business level, e.g. "new credit application received".

What are common to all events are three aspects: form, significance, and relativity to other events (by time, causality, and aggregation). Form means that it is an object with significance meaning that this object is a representation of an activity that happened. In order to specify the activity, the object does consist of attributes describing the activity. Events relate to other events, this can be by time (event A happened after event B), causality (events referring to the same customer ID) or aggregation (event A was caused by the occurrence of events B and C). [7]

A first step towards standardization of events is the common base event (CBE) format. With the standardization of Web Services Distributed Management (WSDM) [10] this standard does specify a WSDM event format. The WSDM event format was implemented within the common event infrastructure (CBI)[1] and is known as the CBE [11]. The CBE does at least require a unique identifier (`globalInstanceId`), a creation timestamp (`creationTime`), an ID of a component which caused the event

[1] "The Common Event Infrastructure (CEI) is IBM's implementation of a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of a wide range of business, system and network Common Base Event formatted events." [11]

(`sourceComponentId`), and data structure identifying the situation that happened (`situation`) [12].

Now with a basic definition of events and the minimal requirements, the following two paragraphs define the events in the respective "event clouds".

4.2 SOA events

Based on the model described in chapter 3, a number of events can be defined that can be useful in monitoring the operation of a SOA.

The first class of events describes the lifecycle of service participants at runtime. Whenever a service participant is started or stopped, a corresponding message can be generated. Further events can be generated when a participant registers itself as provider or consumer for a particular service or service operation.

The second class of events provides insight into the usage of service operations. Each service participant can generate events that provide information about calls to service operations and their results. The level of detail provided by these events can range from aggregated data about a large number of service calls to fine-grained tracking data that allows the monitoring of individual message exchanges as they propagate through the messaging infrastructure. The following levels have been proven to provide practical information:

- An aggregation of statistical data (number of calls, successful completions, and response times) about calls to a particular service operation made or served by a service participant during a defined period of time.
- Information about the result of each individual message exchange.
- Tracking information that details each step of processing of an individual message exchange.

While the first level of detail is mainly useful for accounting purposes and capacity planning, the more fine-grained information provided by the other levels can be correlated with information about business events as detailed below.

The third class of events is concerned with quality of service parameters. Whenever a violation of a QoS parameter defined for a message exchange is noticed by the integration infrastructure, a corresponding event is generated.

Instances of the events discussed above are automatically generated by the SBB library and collected at a central location. Therefore, integration into a corresponding business activity monitoring becomes feasibly without any side-effects for the implementation of business applications using the service infrastructure.

The information contained in the events provides information about the event type, event source (service participant or message exchange), and data specific to the event type. This allows correlation of events within the context of the integration infrastructure. To use this information in the context of business activity monitoring, a correlation between SOA-based identifiers (namely message exchange identifiers) and business activity related identifiers (e.g. a business process identifier) needs to be established. The integration infrastructure provided by SOPware supports this in two ways:

An application using the SBB library to access services can specify an arbitrary identifier as an optional parameter to the API call to invoke a service operation. This identifier is observable in all tracking events that are generated for the message exchange that realizes the invocation.

If this is not possible, e.g. if legacy systems are integrated into the execution of a business process, a correlation message can be generated that connects the message exchange identifier with any business identifier that is part of the message payload and that can be specified by XPath [19] statements.

It is our goal to set the SOA-level events described here in relation to high-level events generated by the business process to get a better understanding of the operation of the whole infrastructure.

4.3 Business and process events

The workflow environment of the use case described in chapter 2 does provide the following CBE events for each invoke or receive in a BPEL process[2]: Entry, Exit, Expired, Failed, Completion forced, Retry forced, Skipped, Stopped, and Terminated.

The events above are possible activities during the BPEL process execution and are CBE. They will be called process events in this paper.

Business events are specified by the messages or events either a business service receives or produces such as "new credit application received".

Because the processes are modeled in BPEL and because they resemble the business process, the process events can indicate business events. In order to denote an event for "new credit application received" the CBE process event "Entry" of the receive process node can be used.

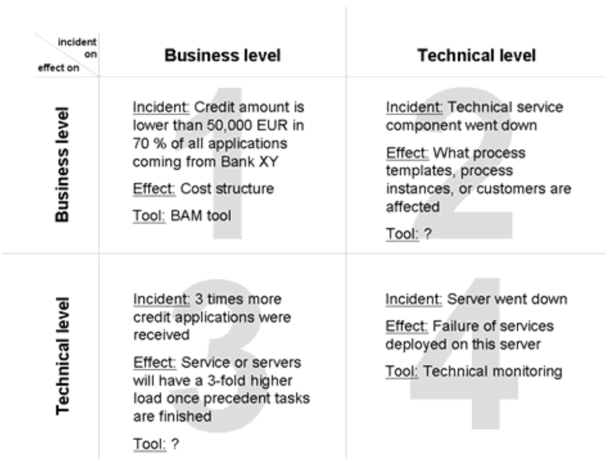
Another way to raise business events is with sensors in BPEL processes as explicit service calls. This was introduced in [16].

5. MONITORING WITH EVENTS

5.1 Use case scenarios

The essential idea behind existing monitoring tools (both business and technical) is to alert when single events happen and then to act upon them [13]. But the humblest low level event can have impact on high level processes and the overall performance of the enterprise as well as high level events can have future impact on IT assets. In this chapter we want to show on simplified examples bases that both "event clouds" can be mapped in order to extract useful information and monitoring results.

[2] IBM WebSphere Integration Developer (WID) 6.0.1 and IBM WebSphere Process Server 6.0



incident on / effect on	Business level	Technical level
Business level	Incident: Credit amount is lower than 50,000 EUR in 70 % of all applications coming from Bank XY Effect: Cost structure Tool: BAM tool	Incident: Technical service component went down Effect: What process templates, process instances, or customers are affected Tool: ?
Technical level	Incident: 3 times more credit applications were received Effect: Service or servers will have a 3-fold higher load once precedent tasks are finished Tool: ?	Incident: Server went down Effect: Failure of services deployed on this server Tool: Technical monitoring

Figure 6. Examples of business and technical incidents and their effects

Figure 6 shows a matrix of four scenarios with business and technical incidents (horizontal) and business and technical effects (vertical). Incidents with an effect on the same level are easy to handle and are state-of-the-art (scenarios 1 and 4).

Scenarios 2 and 3 exceed the scope of the respective monitoring environment and the connections between them are hard to trace.

5.2 Event correlations and patterns

With event correlations and event patterns we will concentrate on scenarios two and three.

Scenario 2: Here we have to detect different activities. 1. That a service component went down or that it is not responding as agreed upon in a service level agreement (SLA). 2. What process instances were using the service component? 3. Which process template are affected, who are the customers, and other details?

For the purpose of illustrating how to write event patterns we will use the straw man event pattern language STRAW-EPL as introduced by [7].

Element	Declarations
Variables	MessageExchange mex, Time timeAt, Operation op, Time timeAgreed, Time actualTime, Correlation c, MessageId mid, CorrelationId cid, BusinessId bid, String processInstanceId, String processTemplateId, Customer cust
Event types	RESPONSETIME_EXCEEDED (mex, op, timeAgreed, actualTime) CORRELATION(c, mid, cid, bid) new_credit_application.Entry(processInstanceId, processTemplateId, cust)
Rel. operators	**and**
Pattern	RESPONSETIME_EXCEEDED **and** CORRELATION **and** new credit application.Entry
Context test	mex.getId=mid **and** bid=processInstanceId
Action	**create** BamScenario2Event("Scenario 2", …)

Figure 7. Event pattern for scenario 2 in STRAW-EPL

Here, two SOA events are correlated with a business event. Once the response time of a component is exceeded we take the correlation event which holds both the link to the service component and the business process instance data and correlate them. If the pattern matches a new event will be created.

We suggest the `BamScenario2Event` and `BamScenario3Event` to be a CBE and to be sent via the ESB to the BAM tool. The ESB can then transform it to proprietary event formats (e.g. "ARIS PPM Event Format") or pass it on to other CBE consumers. With this approach BAM tools could be loosely coupled just like services in a SOA.

Scenario 3: In scenario 3 we receive more credit applications than normal. If the credit amount is above 50,000 Euros, the process does start a human task and waits for the task to be finished before proceeding (which according to the BPEL is a service call). Before completing the manual tasks, no preceding service call is made, meaning that the SOA framework has no way of knowing about the imminent increase in service calls. Therefore, no pro-active measures, like the deployment of additional service providers, can be taken.

In order to measure the number of credit applications with events over a period of time we have to extend the STRAW-EPL with a sequence of events (identifier "sequence") with an index *i* and an identifier for a time window ("within").

Element	Declarations
Variables	String ProcessInstanceId(i), String ProcessTemplateId(i), Customer cust(i)
Event types	new_credit_application.Entry (processInstanceId(i), processTemplateId(i), cust(i))
Rel. operators	**and**
Pattern	**sequence** (new credit application.Entry(i)) **within** last 8 h
Context test	processTemplateId(i)=processTemplateId(i+1)
Action	**create** BamScenario3Event(…)

Figure 8. STRAW-EPL for scenario 3

With the creation of `Scenario3Event` and the business process knowledge the proceeding service calls (and thus the IT assets affected) are automatically identified.

Again, the event patterns above do not claim to be real world patterns. They do only demonstrate that with event patterns and EP a technical event cloud can be matched with a business event cloud.

5.3 BAM

Although the ARIS PPM tool does only meet the first two features of the BAM event processing checklist in [14] (i.e. real time event data computation and single event triggering), it is widely used and does use the EPC process notation. This does mean that a business process lifecycle can be reached from process modeling and process automation to monitoring and process re-design and so forth with only one business process notation.

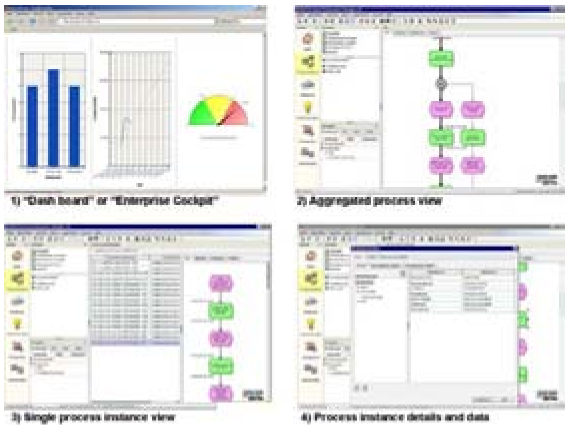

1) "Dash board" or "Enterprise Cockpit"
2) Aggregated process view
3) Single process instance view
4) Process instance details and data

Figure 9. Different ARIS PPM BAM views

The functionality of the remaining three features of the BAM event processing checklist (event streams processing, event pattern triggering, and event pattern abstraction) would be met by implementing scenarios 2 and 3. What is still missing is a monitoring user interface to visualize the results of our EP scenarios.

The possibility we suggest is to use an existing BAM tool such as ARIS PPM. Since those tools are at least capable of single event processing and the visualization of single events such as `BamScenario2Event` or `BamScenario3Event`.

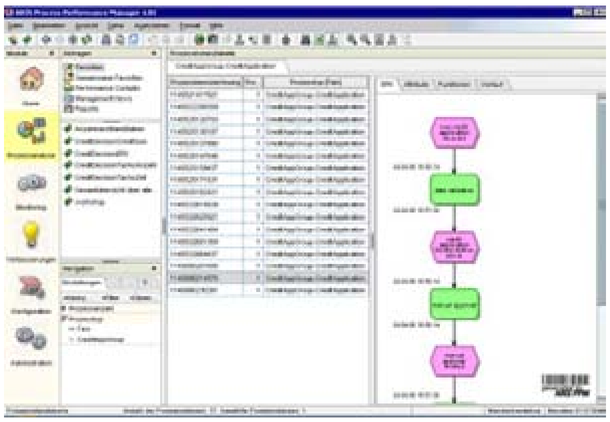
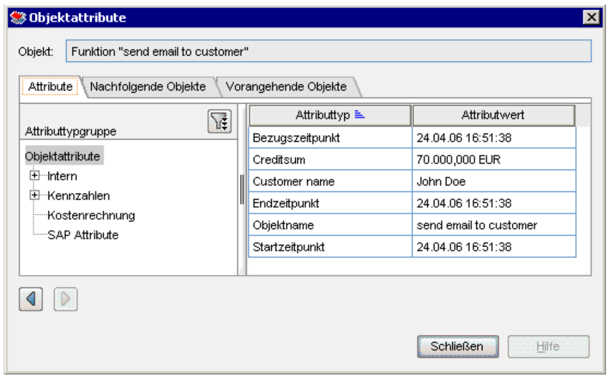


Figure 10. Scenario 2 visualization

Figure 10 shows a list of process instances where the pattern (see Figure 7) matched. Every process instance can be shown in the appropriate EPC process model exactly as it was executed. The single process nodes contain the business data monitored. This is where further information about the process, customer, or other details can be retrieved as shown in Figure 11.



Objektattribute

Objekt: Funktion "send email to customer"

Attribute | Nachfolgende Objekte | Vorangehende Objekte

Attributtypgruppe

Objektattribute
- Intern
- Kennzahlen
- Kostenrechnung
- SAP Attribute

Attributtyp	Attributwert
Bezugszeitpunkt	24.04.06 16:51:38
Creditsum	70.000,000 EUR
Customer name	John Doe
Endzeitpunkt	24.04.06 16:51:38
Objektname	send email to customer
Startzeitpunkt	24.04.06 16:51:38

Figure 11. Details of a process node in ARIS PPM

The visualization of scenario 3 shows a speedometer like diagram with the number of process instances within a time window. Figure 12 shows that the number of running processes has reached a critical area. Due to the coupling of the BAM EPC view with the EPC process models all following activities and service calls of a process template or instance can be identified.

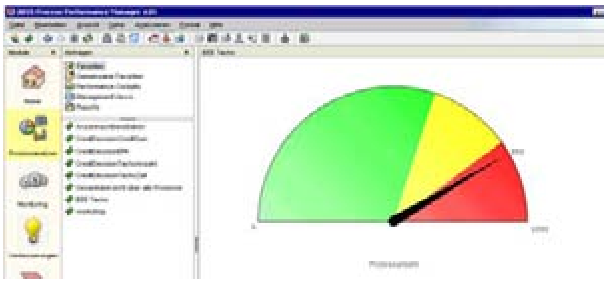


Figure 12. Scenario 3 visualization

6. CONCLUSION

EP as a very young discipline has not yet reached a level of agreed standards, wide spread tools, or good market penetration. It is our belief that it holds the potential to leverage synergies between the areas of BPM (management, modeling, and automation of processes), BAM (monitoring of processes) and SOA monitoring and management, among others.

Our study shows one possible way of utilizing the capabilities present in the technologies mentioned above to achieve this goal. We showed links between events (both business-events and events in an SOA), information processing, and processes in order to leverage modern application and monitoring platforms such as SOPware and ARIS PPM. As theoretically introduced in [20] our study shows furthermore that monitoring using EP in combination with BPEL processes provides a comprehensive and agile foundation for business process optimization that ensures efficient and flexible business processes.

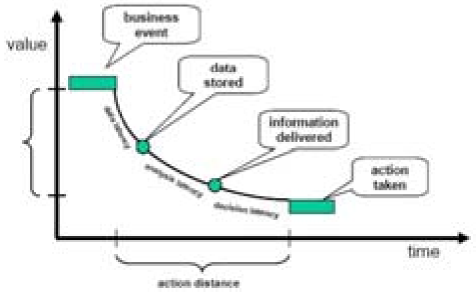


Figure 13. "Business Value of Data Freshness" [15]

To illustrate a business value behind our study we want to refer to the time-value curve of [15] where the author shows the connection between "data freshness" and the "business value". This "data freshness"[3] we can provide by applying EP in order to map the clouds of events and to complete functionalities today's BAM environments are not yet able to provide.

7. REFERENCES

[1] J.B. Hill, J. Sinur, D. Flint, M.J. Melenovsky, Gartner's Position on Business Process Management, 2006, Gartner Research, 16 February 2006, pp. 5-8.

[2] P. Küng, C. Hagen, M. Rodel, S. Seifert, Business Process Monitoring & Measurement in a Large Bank: Challenges and selected Approaches, Proceedings of the 16th International Workshop on Database and Expert Systems Applications, IEEE Publications, 2005.

[3] R. Davis, Business Process Modelling with ARIS. A Practical Guide, Springer, London, 2001.

[4] OASIS Web Services Business Process Execution Language (WSBPEL) TC, Organization for the Advancement of Structured Information Standards (OASIS), http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, accessed on 17 April 2006.

[5] B. Gassman, Who's Who in Business Activity Monitoring, 4Q05, Gartner Research, 12 April 2006, p. 3.

[6] J. Deeb, F. Knifsend, Using BAM to Empower the Organization: Unifying Events and Processes, http://www.BIJOnline.com/index.cfm?section=article&aid=130, 2005.

[7] D.C. Luckham, The Power of Events, An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley, Boston, 2002.

[8] Event Stream Processing: A New Computing Model, http://www.eventstreamprocessing.com, accessed on 15 April 2006.

[9] R.W. Schulte, The Growing Role of Events in Enterprise Applications, Gartner Research, 9 July 2003, pp. 2-3.

[10] Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1, OASIS, 9 March 2005, http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf, accessed on 19 April 2006.

[11] Common Event Infrastructure, Intelligent automation that saves time and improves resource utilization, IBM, http://www-306.ibm.com/software/tivoli/features/cei, accessed on 19 April 2006.

[12] Autonomic Computing Toolkit, Developer's Guide, IBM, August 2004, http://www-128.ibm.com/developerworks/autonomic/books/fpy0mst.htm#HDRAPPA, accessed on 19 April 2006.

[13] D.C. Luckham, Why Business Activity Monitoring Must Use CEP, 21 July 2004, http://complexevents.com/?p=23, accessed on 20 April 2006.

[14] D.C. Luckham, Can We Bring Some Order to the Business Activity Monitoring Space?, 16 June 2005, http://complexevents.com/?p=5, accessed on 20 April 2006.

[15] R. Hackathorn, Current Practices in Active Data Warehousing, November 2002, http://www.dmreview.com/whitepaper/WID489.pdf, pp. 23-24, accessed on 21 April 2006.

[16] D. Jobst, T. Greiner, Modern Process Management With SOA, BAM und CEP, From static process models to executable workflows and monitoring on business level, 1st Event Processing Symposium, Hawthorn NY, March 2006, http://complexevents.com/wordpress/slides/CITT_CEPSymposium_Jobst_Greiner.pdf, accessed on 21 April 2006.

[17] World Wide Web Consortium, Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, March 2006, http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327/, accessed on 24 April 2006.

[18] J. Belger, Die service-orientierte Platform der Deutschen Post, JavaSPEKTRUM 1/06, p. 22.

[19] XML Path Language (XPath) Version 1.0, W3C Recommendation, 16 November 1999, http://www.w3.org/TR/xpath, accessed on 24 April 2006.

[20] J. Deeb, What Does Complex Event Processing Do for BPM, 13 June 2005, http://www.bpm.com/FeaturePrint.asp?FeatureId=171, accessed on 24 April 2006.

[21] B. Gassman, Gartner Study Reveals Business Activity Monitoring's Growing Value, Gartner Research, 18 April 2006

[3] According to the latest Gartner research today's latency requirements between business events and notification is less than 15 minutes. This is likely to decrease to less than 60 seconds over the next 6 years [21].

Business Activity Monitoring of norisbank Taking the Example of the Application easyCredit and the Future Adoption of Complex Event Processing (CEP)

Torsten Greiner, Willy Düster, Francis Pouatcha
norisbank AG
Rathenauplatz 12-18
D-90489 Nürnberg
torsten.greiner@norisbank.de

Rainer von Ammon, Hans-Martin Brandl, David Guschakowski
CITT GmbH / TietoEnator GmbH
Konrad-Adenauer-Allee 30
D-93051 Regensburg
rainer.ammon@citt-online.com

ABSTRACT

The kernel business process of easyCredit of the norisbank as a German Online Credit System is described with all its detailed process steps. Business Activity Monitoring is a basic condition for the successful operation of the system. The As Is-solution is faced with the future concept based on CEP/ESP. For that reason the process has to be redesigned in the sense of SOA and flexibly implemented by means of a BPEL-based Workflow Engine. In this connection the events of a BPEL-engine are differentiated from the events, which are being processed in a so called "event cloud" by a CEP system for a real time BAM. The possibilities and advantages of a real time able BAM are being shown taking the example of the easyCredit.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: l*anguages*

General Terms
Performance, Design, Standardization, Languages, Theory

Keywords
Business Activity Monitoring, Complex Event Processing, Event Stream Processing, Business Process Management, Service Oriented Architecture, online instant credit system

1. INTRODUCTION

Business processes as a whole realize the business strategy of an enterprise in a save, correct and economic way. Thereby IT systems support and optimize the performance of the business processes. In the example of the browser based easyCredit application (rated credit with online instant confirmation on the Internet) of the norisbank AG is described a potential business process monitoring from an economic view based on key figures. This economical monitoring of the business processes on the base of key figures is called Business Activity Monitoring (BAM) [10][11].

The norisbank has realized BAM by a pipeline model. This model assumes that each credit application runs through several processing steps within its entire life cycle. The single steps can be imagined as a production line or a pipeline. In this pipeline model we can trace how many contracts respectively applications are located on which position of the pipeline. The actually traced values of a pipeline section can be compared with threshold ranges, so that in the case of significant deviations indicating a technical fault in the process an alarm can be raised.

In the following the essential basics for the realization of the business process monitoring of the rated credit application easyCredit with BAM is introduced. After that the real system is described, how it has successfully been implemented in the norisbank AG. At last it is shown, how new paradigms of Complex Event Processing (CEP) [9] will be used in the next generation of the easyCredit for realizing an intelligent, real time able BAM platform.

2. SOME REMARKS ABOUT THE BACKGROUND OF EASYCREDIT

The norisbank AG - a 100% affiliate of the DZ-Bank AG since October 1st, 2003 – runs about 100 branches in Germany. Their kernel business is the allocation of consumer credits. In April 2000 norisbank was the first bank in Germany to offer a rated credit application on the Internet with online instant confirmation. This application was certified by the TÜV (Technical Controlling Association, MoT) in 2003 as the world wide first bank product.

Due to the change of the ownerships at the norisbank AG it became necessary to develop a new fully automated and mass business capable rated credit application (easyCredit[1]). A technical relaunch, as well as an economical further development of the old application was accomplished.

In 2002 the norisbank introduced a new technical platform as a basis for a modern Internet branch bank [7]. The technical architecture of this platform was realized by the operation system

[1] http://www.easycredit.de

Solaris of SUN. This platform provides a scalable architecture, based on the standard of Java Enterprise Edition (Java EE). The Java EE model defines the standard for the implementation, configuration and for operating distributed applications. Furthermore Java EE is web-based, i.e. it is assumed that the clients (webbrowser capable systems) are interacting with the application via the protocol http or https. Thus redevelopment of the easyCredit application was made by the Java EE technology as well. This redevelopment, too, was certified in 2004 by MoT.

Meanwhile about 900 partner banks, with about 12.000 distribution agencies and about 32.000 branch users are using this application, but also the same amount of external customers from the Internet.

From the view of the end users (partner bank, Internet) it's not only of interest, whether the application and the process are available, but the cycle time of a contract within the norisbank is essential, too. The cycle time is here defined by the total processing time from the reception of the credit application until the payment onto the account of the customer. For this reason it is especially important to know, which contract has which actual status.

It is the aim of business process monitoring within a pipeline model to define traffic light status for each pipeline section which has to be controlled:

Green = amount of contracts in a pipeline section are in the defined interval

Yellow = amount of contracts in a pipeline section are approaching the defined interval limits

Red = amount of contracts in a pipeline section are outside the defined interval

Only in the case of an alert in time the throughput of the contracts can be constantly maintained with added effort of personal resources.

3. BUSINESS ACTIVITY MONITORING

For the realization of Business Activity Monitoring the understanding of the business processes are indispensable. In this connection the kernel business process of the easyCredit is to be understood in the way that the process is being started by the credit application of an external or internal customer and only ends with the delivery of an agreed result to the customer (end user). In the following only the business process steps, which are relevant for the monitoring of the application way "Internet guest", are shown and after that the realization of the BAM on the basis of the pipeline model are explained.

In this process a customer uses the Internet to get a credit offer. At first the customer gets to the "mini calculator" page. There he can choose diverse credit parameters like the amount of credit, duration, and optionally an insurance product. After that he will automatically be given the adequate interest rate as well as the monthly amount of interests. Has the customer decided to continue with these conditions, he has to enter his household data in the next step. These data include the monthly income and expenses as well as credit contracts, which may have to be repaid. In the following the customer has to enter his personal data inclusive his address. After that the employment data form has to be processed. As a next step the customer has to enter in the account data page, whereto the money shall be transferred and from which account the monthly rates shall be collected. All the entered data are checked by appropriate procedures for pausibility which will not be explained here in detail.

Does the customer continue with the application process, the credit application with all its data entered so far will be stored in the system. From this point the control of this application within the monitoring system is possible (see section B.1 in fig. 1). Only now an inquiry at the credit agencies Infoscore and Schufa will be made. According to the results the decision on the credit (CD) will be met. The result of the CD will be arranged as follows:

CD red: The customer will be briefly indicated that he cannot get a credit offer (e.g. because of insufficient creditworthiness). For the customer it is the end of the offer process. In the background however a further process automatically produces a letter of denial in the form of a PDF-document. Another process provides the generated PDF-document as a print job. All print jobs are being gathered, printed and sent off in a cyclic way in another process.

CD grey: The customer gets a non-binding offer on the screen assuming a averaged good creditworthiness, as there are no informations of the Schufa in this case. The customer will be informed that he can expect a written offer within short time. Parallel to this the resubmission "reviewing" is set in the background. That means that the further processing of the credit application will be continued by an employee of the norisbank.

CD green: In this case it is differentiated, whether the customer has entered the credit contracts, which will have to be repaid to norisbank. In this case an offer is shown to the customer on the screen and he will be informed that he will get the offer documents by post. In addition the resubmission "reviewing" is set.

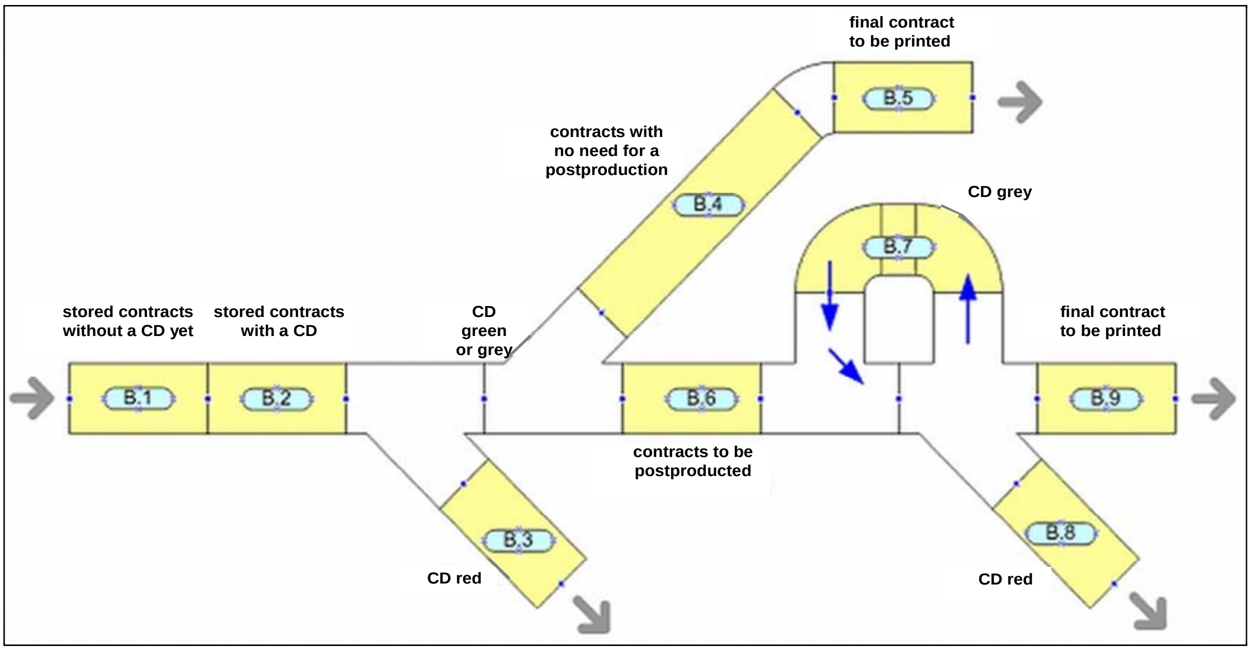
The result of this description is the simplified shown life cycle model of the application way "Internet guest" (see fig. 1).

4. REALIZATION OF THE PIPELINE MODEL

A pipeline model is used for the technical monitoring of the business processes by norisbank. This model assumes that every credit application within its total lifecycle runs through several processing steps, like in a production line or in a pipeline. In this pipeline model we can trace how many contracts respectively applications are located on which position of the pipeline.

Precondition for this is that each change of a status of an application is recorded with a corresponding time stamp in a database. The actually traced values of a pipeline section are then compared with the defined threshold ranges, so that in case of significant deviations indicating a technical fault in the process an alarm can be raised.

In order to check any faults in the business processes the following monitoring principles within the pipeline model will be used:



Legend:
B.1: stored applications, for which no credit decision (CD) was made yet.
B.2: stored applications, for which a credit decision was made already.
B.3: applications, for which the last credit decision was red.
B.4: applications, with CD green, which don't have to go into reviewing.
B.5: applications, with CD green, which don't have to go into reviewing,
the contract of which is ready for printing.
B.6: applications, with CD green, which are being reviewed, for which however there hasn't been done any CD in the reviewing yet.
B.7: applications with CD grey in the reviewing.
B.8: applications with CD red in the reviewing.
B.9: agreed applications, the contract for which is ready for printing.

Figure 1: Simplified shown life cycle model of the application way "Internet guest"

- Finding the amounts in the single pipeline sections,
- Monitoring, whether the applications are moving forward in the pipeline,
- Checking, whether all applications/contracts within a certain time frame have reached a final status.

For this the following monitoring variants are used in detail:

Pipeline-snapshot: Establishing the absolute amount of applications within the single pipeline sections at defined times (e.g. each hour).

Pipeline-progress: Establishing, which application objects have newly reached the corresponding pipeline section within a defined time interval (e.g. each 15 minutes). The absolute amount of applications is established.

Application specific finalising: Here it is checked, whether all applications within a certain time interval have reached the defined final status (e.g. one working day per each application). Defined final states are the sections B.3, B.5, B.8, B.9.

Wait time-monitoring: Establishing, how much time it takes for an application from one pipeline section to the next respectively whether an application stays for too long in a certain pipeline section. Critical pipeline sections are: B.2, B.5, B.6, B.7, B.9.

The monitoring of the single application states is done by cyclical database requests. Hereby the time stamp of the corresponding status entry in the database serves as a reference. E.g. if the Schufa credit agency is not available, it is easily possible to recognize through evaluating the pipeline progress that the amount of applications with status CD grey have been raised significantly.

5. FURTHER DEVELOPMENT OF THE MONITORING PROCEDURE WITH CEP/ESP

The so far described approach assumes that each credit application with its total life cycle runs through several processing steps, like in a production line or within a pipeline. Precondition for a corresponding technical monitoring is however that each status change of an application is recorded with a corresponding time stamp in a database. Only in this way a simple database request can establish, how many contracts are located in which place within the pipeline. The advantage of this procedure is the simple realization.

The disadvantage of this approach however is that only a frozen state of an existing constellation at time point t of the data base request can be seen. A real time monitoring of the economical states of the application is only possible, if status changes of applications are requested in the data base permanently. This means: if the answer for a request is wanted each second, the request has to be entered each second as well. This solution cannot be realized for hundreds of different requests. Therefore the database is a bottleneck in the case of complex status changes, which shall be monitored in real time. Also the triggers, which were integrated later in traditional databases, don't solve the problem, as these don't scale.

Event Stream Processing (ESP) or Complex Event Processing (CEP) are paradigms, which are helpful to react in real time to changes of states by corresponding informations [9]. With CEP/ESP messages, informations or data are correlated, aggregated, analyzed and evaluated in real time. These newly generated informations then provide the base for further decisions. Thus a CEP/ESP-platform becomes an intelligent BAM-tool, which also offers the possibility of dynamic visualization. In a next step this pipeline model shall be realized with methods of CEP/ESP.

5.1 Redesign as SOA and Use of a BPEL-based Workflow Engine

For the new approach of easyCredit it is necessary to redesign the architecture of the system in the sense of a SOA [3][16]. As a principal difference to EAI [12] a SOA is based on the business processes (see fig. 2). At each process step another enterprise internal or external process or a service respectively a software component can be called and eventually a change in a database or in a legacy system, e.g. in an ERP-system like R/3 or in a CRM-system like Siebel, can be caused. Fig. 2 shows the example of an online-credit-system that this architecture could arbitrarily cascade [1]. All software-components respectively services are defined e.g. by the Web-Service-Definition-Language (WSDL) and are bound as web-service to a process step.

However hereby it has to be guaranteed that for the aim of a real time BAM no performance problems will be caused by still relatively heavy and slow XML-based protocols like SOAP [15] because of longer latency times.

This also goes for the use of a BPEL-based workflow engine [13]. Though the business process can be standardized and flexibly implemented in this way and even be modified at run time of the system on a high level, i.e. by the means of workflow design tools by non-IT-experts, if applicable even directly by a business department, for the realization of new business and marketing strategies. On the other hand numerous, concurrent business process instances could cause performance and scalability problems of the BPEL-engine. This has to be considered in time towards the required real time performance of the BAM at the dimensioning of the system and the resource planning (sizing-project).

5.2 Events for the Workflow Engine Versus Events for CEP/ESP

The business process is controlled e.g. in the sense of an eEPC-notation [5] of events, like “credit application received”, “credit application checked for completeness”, “Schufa-information

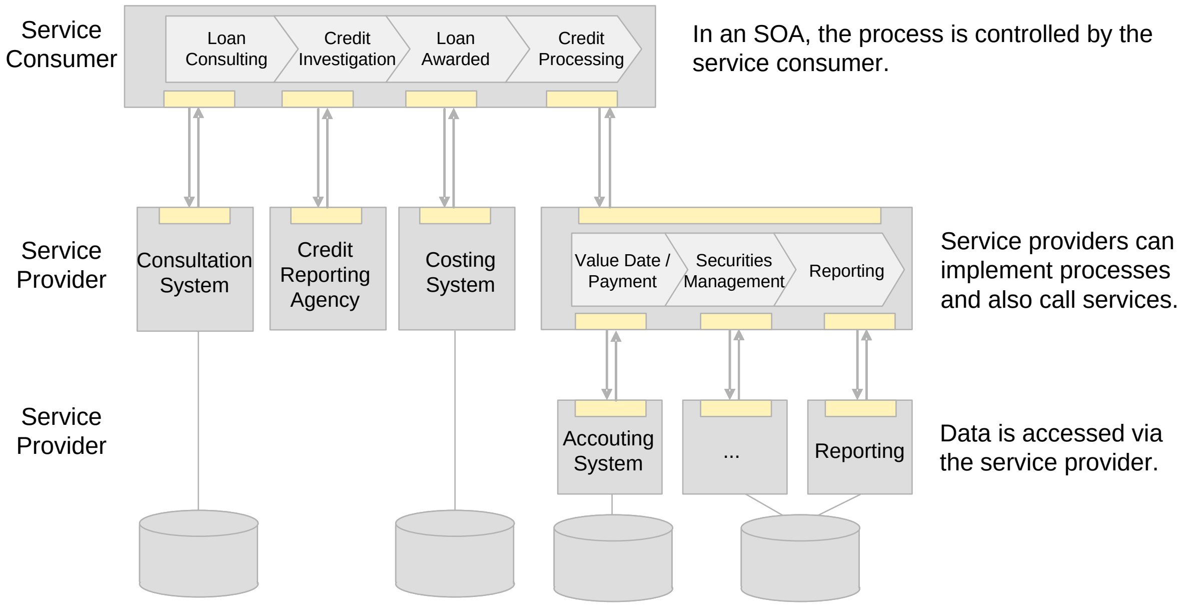


Figure 2: The SOA Challenge - design horizontal and vertical coupling of services

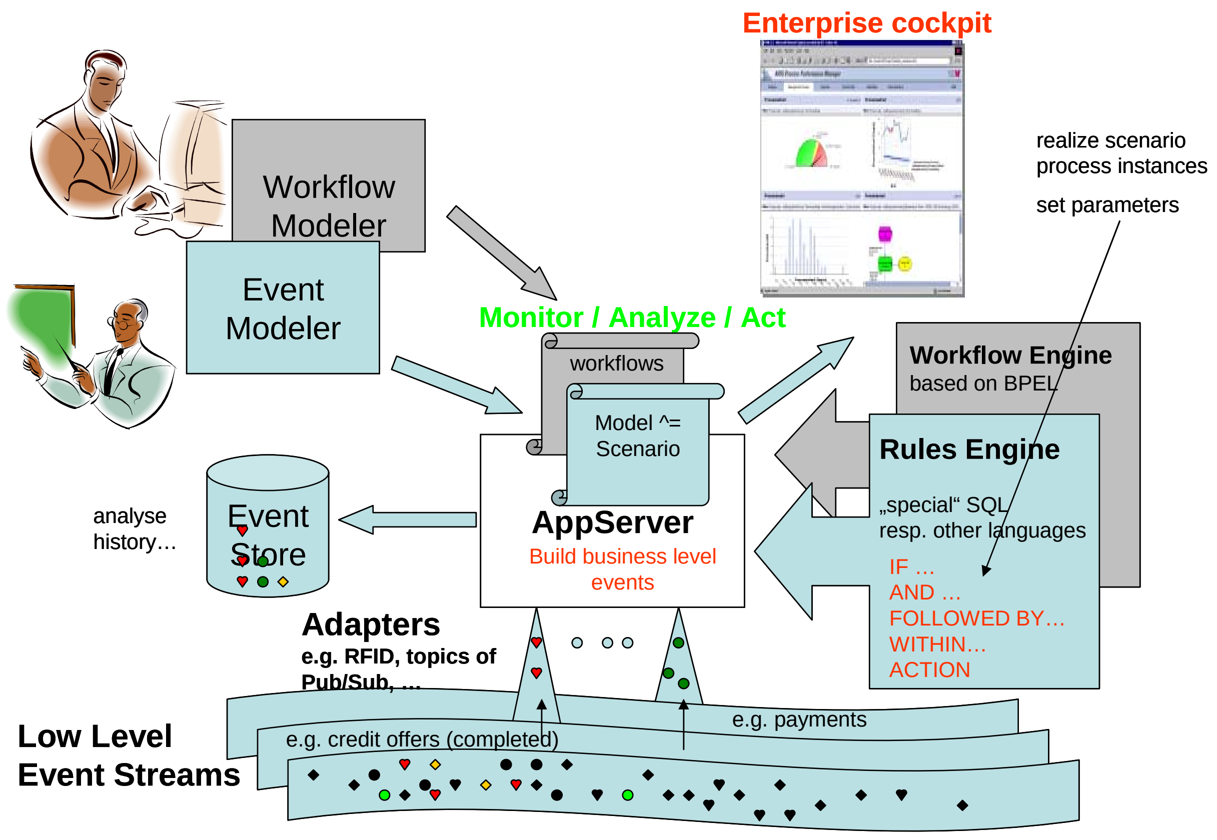


Figure 3: The technology challenge and the principle of BPM/BAM/CEP, e.g. for "Next Generation Instant easyCredit System"

received" and so on. These events are manually caused by men, e.g. by an employee or by actions of the system. These events are needed by the workflow engine for flow controlling. These events, however, are no events, which are filtered from the event cloud or from an event stream by a CEP- or ESP-system and processed, e.g. for the visualization in a BAM. It is the job of a business process modeler to design the process model by interviewing the process-owner or the business department as a chain of events and actions in such a way that the process model can be executed by a BPEL-based workflow-engine (see fig. 3). These thousands and presently archived eEPC-models at banks and other users are at present not designed as "executable workflows" and have to be redesigned more fine grained – also under SOA-prospects.

The Business Process Management System (BPMS) (see [14]) generates at executing the business process instances partly autonomously BAM- respectively CEP-relevant informations, e.g. time stamps for each single process step, whereby in the BAM throughput times can be monitored and analyzed. Additionally specific events can be generated through appropriate implementation of actions, e.g. for those in chapter 4 mentioned monitoring-views "establishing the absolute amount of applications within the single pipeline sections at defined times", "establishing, which application objects have newly reached the corresponding pipeline section within a defined time interval", or the calculation of a credit sum or of the interest rate. This can be realized in the Java EE environment via JMS as the Publish-/Subscribe-method or with CORBA analogically via the "notification service" and so on.

Altogether there can be on the network level a very large amount of events of different types in a certain time window (1 minute, 1 hour, 1 day and so on) – metaphorically as an unordered "event cloud" or transformed as a chronologically ordered "event stream".

The event modeler (see fig. 3) decides, which of these events have to be filtered for which BAM-view from one or more streams, if necessary have to be aggregated and correlated as higher business level events (see chap. 4, e.g. pipeline progress, wait time monitoring) and in which time window these events have to be held and stored (see in detail [9]). According to CEP/ESP-systems, just entering the market, those event scenarios again can be generated very quickly and modifiable at any time by means of special high level tools without IT experts.

There are special, often already prebuilt adapters for the filtering of each event type (see. fig. 3). Examples are SNMP-, email- and log file-adapters for searching for specified strings. The aggregation and correlation take place according to the event scenarios through the used CEP/ESP-system by means of their Event-Processing Language (EPL). The CEP-discipline, just coming up, is presently discussing - still controversial -, how an appropriate language or an appropriate user interface for this task shall be designed. At present the first CEP-platforms offer SQL-like languages, which, however, are extensively enhanced, and process events - precompiled and "in memory" – in a highly performing way (e.g. the "Event Query Language" (EQL) of the Open Source Platform Esper [6] or the "Continous Computational Language" (CCL) of Coral8 [4]). Other systems provide an user interface without any programming as far as possible (e.g. APAMA [2]).

In the future an important, new task will be the modeling of appropriate event patterns by the new role of the event modeler.

6. Conclusion

Though from the perspective of the next generation easyCredit the BAM-views have to be defined first, which are to be monitored – if necessary in real time –, as well as all required actions and alerts have to be determined. After that the necessary event types on the network level (in the event stream) have to be decided and aggregated or correlated in appropriate models. Furthermore it will be evaluated, which BPEL-engine shall be used and which events of which actions in the credit process will have to be generated. For the event management and processing in real time diverse CEP-/ESP-systems are being evaluated.

7. REFERENCES

[1] Ammon, R.v., Pausch, W. and Schimmer, M. Realisation of Service-Oriented Architecture (SOA) using Enterprise Portal Plattforms taking the Example of Multi-Channel Sales in Banking Domains, *Wirtschaftsinformatik 2005*. Ferstl et al. (Publ.), Heidelberg, Physica-Verlag 2005, 1503-1518.

[2] Apama, http://www.progress.com/realtime/products/apama/apama_technology/index.ssp.

[3] Brandner, M. et al., Web services-oriented architecture in production in the finance industry, *Informatik Spektrum*, Volume 27, No. 2, 2004, 136-145.

[4] Coral8, http://www.coral8.com/, downloaded 2006-04-24.

[5] eEPK – erweiterte Ereignisgesteuerte Prozesskette, http://de.wikipedia.org/wiki/Erweiterte_ereignisgesteuerte_Prozesskette, downloaded 2006-04-24

[6] Esper, http://esper.sourceforge.net, downloaded 2006-04-24.

[7] Greiner, T. Lachenmayer, P. Bereitstellung einer neuen technischen Plattform als Grundlage für eine moderne Internetfilialbank, *Banking and Information Technology*, Institut für Bankinformatik und Bankstrategie an der Universität Regensburg, Regensburg, 2002, 53-61.

[8] Greiner, T., Düster, W. Monitoring von Geschäftsprozessen mit OpenSource Produkten aus Endkundensicht, *Banking and Information Technology*, Institut für Bankinformatik und Bankstrategie an der Universität Regensburg, Regensburg, 2005, 49-61.

[9] Luckham, D., *The power of events*, Addison Wesley , Boston, San Francisco, New York et al., 2002.

[10] Luckham, D. The Beginnings of IT Insight: Business Activity Monitoring http://www.ebizq.net/topics/bam/features/4689.html, 2004, downloaded 2006-04-24

[11] Kochar, H. Business Activity Monitoring and Business Intelligence http://www.ebizq.net/topics/bam/features/6596.html, 2005, downloaded 2006-04-24

[12] Meinhold, G. EAI und SOA: Die Komponenten fallen nicht vom Himmel, *Objektspektrum*, No. 2, 2004, 33-36.

[13] OASIS, Web Services Business Process Execution Language Version 2.0, Draft, Dec. 2005, http://www.oasis-open.org/committees/download.php/16024/wsbpel-specification-draft-Dec-22-2005.htm, downloaded 2006-04-24.

[14] SixSigma, BPMS, http://www.isixsigma.com/dictionary/BPMS-536.htm, downloaded 2006-04-24.

[15] SOAP, http://de.wikipedia.org/wiki/SOAP, downloaded 2006-04-24.

[16] Woods, D., *Enterprise Services Architecture*, O'Reilly, Gravenstein, 2003.

Author Index

ISBN: 3-939352-05-5
ISBN: 978-3-939352-05-1