UNIVERSITÄT
MANNHEIM

System Support for Proactive Adaptation

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
der Universität Mannheim

vorgelegt von

Sebastian Jason VanSyckel

aus Nürnberg

Dekan: Dr. Jürgen M. Schneider
Referent: Prof. Dr. Christian Becker
Korreferent: Prof. Dr. Arkady Zaslavsky

Tag der mündlichen Prüfung: 12. Juni 2015

Abstract

Applications in our modern, pervasive computing environments have to adapt themselves or their context in order to cope with changes. In the process, these pervasive applications should be as unobtrusive as possible, i.e., their adaptation should be automatic. In dynamic multi-user systems with shared resources and interactive applications, such adaptations cannot be scripted in advance. Instead, they have to be calculated at runtime. However, the necessary calculations quickly exceed the complexity that can be handled in real-time, i.e., without causing significant delays. The concept of proactive adaptation allows to change applications and/or context based on prediction of context and user behavior. Hence, proactive adaptation can reduce adaptation delays and avoid context interferences by determining coordinated adaptation plans ahead of time, instead of reactively when adaptation becomes necessary. Further, it helps to provide a seamless service to the user, while optimizing the overall system utility.

This thesis presents a general framework and middleware-based system support for coordinated proactive adaptation in dynamic multi-user pervasive systems. The framework consists of five major components. The context interaction model and corresponding context broker offers context information, prediction, as well as actuation in a uniform fashion. The application configuration model allows applications to specify their requirements towards their context, as well as detail user preferences and duration-dependent utility and cost functions for adaptation optimization. Configuration algorithms calculate and rate all adaptation alternatives of an application given a current or predicted context and the specified rating functions, before coordination algorithms find interference-free adaptation plans for situations in which multiple applications share a context space. Finally, the adaptation control component combines the individual components of the framework in a two-dimensional control loop for proactive and fallback reactive adaptation. The prototype framework is evaluated in real-time simulations of an interactive pervasive system using recorded user traces.

Acknowledgments

I would like to thank my advisor Prof. Dr. Christian Becker for his continuous support, encouragement, and mentoring since I became a diploma candidate at his chair back in 2010. Christian, thank you for giving me the freedom to follow my own drum – whenever possible – and making the last five years such a fun ride, whether at the chair or on the other side of the world. Gin & Tonic and Dim Sum are good times.

I would like to thank Prof. Dr. Arkady Zaslavsky for his input and his willingness to act as the second reviewer, as well as Prof. Dr. Alexander Mädche for finding the time to join the board of examiners on short notice.

I would like to thank all the people I had the pleasure of working with throughout the years, namely Janick Edinger, Kerstin Goldner, Dr. Florian Heger, Laura Krammer, Christian Krupitzer, Markus Latz, Jens Naber, Dr. Verena Majuntke, Felix Maximilian Roth, Dominik Schäfer, Prof. Dr. Gregor Schiele, and Dr. Richard Süselbeck. I learned a lot from the *first generation* and try to pass on their knowledge as best I can. To Gregor, thank you for teaching me science, as well as the nightlife of Galway. To the *second generation*, thank you for the great camaraderie. Good company makes long nights in the lab much more enjoyable.

Last but not least, I would like to thank my family and friends for always being there for me. To my mother Waltraud, thank you for raising me to be curious and open-minded. To my sisters Anja and Alisa, thank you for having my back. To my partner Tina, thank you for motivating me, challenging me to be better every day, and being my tower of strength.

Acknowledgments

This work was supported by the German Research Foundation (DFG) under grant BE 2498/7-1 “Proaktive Adaption in ubiquitären Systemen”. I would like to thank each thesis student and research assistant involved in the project for their contributions, namely Hai Son Dang, Christine Frank, Dominik Schäfer, Irina Toncheva, and Michael Waleczek.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Programming Interface
BFS	Breadth-First Search
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DFS	Depth-First Search
DNF	Disjunctive Normal Form
GB	Gigabyte
GHz	Gigahertz
HCI	Human-Computer Interaction
I/O	Input/Output
IT	Information Technology
kB	Kilobyte
MB	Megabyte
PC	Personal Computer
RAM	Random Access Memory

AC	Adaptation Control
CB	Context Broker
CM	Configuration Management
SM	Situation Management

Q^{AC}	Context Adaptation Capability Query
Q^{AI}	Context Adaptation Instruction Query
Q^{L}	Context Location Query
Q^{S}	Context State Query
Q^{T}	Context Time Query

CC ..... Context Contract
CI ..... Context Influence
IS ..... Interference Specification

CSIS ..... Context Service Index Structure
E-DFS ..... Exhaustive Depth-First Search
E-DFS w/ CSIS .. Exhaustive Depth-First Search with Context Service Index Structure
E-DFS w/o CSIS . Exhaustive Depth-First Search without Context Service Index Structure
O-DFS ..... Ordered Depth-First Search
O-DFS w/ CSIS .. Ordered Depth-First Search with Context Service Index Structure
O-DFS w/o CSIS . Ordered Depth-First Search without Context Service Index Structure

BT ..... Backtracking
CBJ ..... Conflict-Directed Backjumping
FC ..... Explicit Forward Checking
FC-CBJ ..... Explicit Forward Checking with Conflict-Directed Backjumping
O-BT ..... Optimal Backtracking
O-CBJ ..... Optimal Conflict-Directed Backjumping
O-FC ..... Optimal Explicit Forward Checking
O-FC-CBJ ..... Optimal Explicit Forward Checking with Conflict-Directed Backjumping

1 Introduction

This chapter introduces the present thesis with a motivation, the statement of its research questions and contributions, and gives an overview of its remaining structure. Afterwards, Chapter 2 discusses the theoretical background.

1.1 Motivation

In the modern computing landscape, a multitude of connected computational devices, ranging from embedded sensors over smartphones up to all-purpose computers, form interactive, IT-augmented environments. In such *pervasive environments* or *systems*, applications can adapt to changes in the environment, such as physical conditions, available computing resources and services, user-related information, or simply any information deemed relevant for their situation – their so-called *context* – in order to provide a better service to their users. Adaptation frameworks, such as 3PC [46], Aura [39], and Gaia [92], support these *context-aware applications* by providing access to context information, mediating between applications and resources, as well as assuming responsibilities with regard to adaptation calculation and control.

Typically, adaptation in pervasive systems is *reactive*, i.e., the adaptations are determined and executed after a change in an application's context forces a reaction. *Proactive adaptation* allows to change applications and/or their context based on prediction of context and user behavior, reducing adaption delay and frequency in order to provide a seamless service to the user. Especially dynamic multi-user systems, in which adaptations to certain situations can not be scripted, are application scenarios that can benefit from proactive adaptation. Examples are interactive workspaces at universities, or smart environments in public spaces like libraries, hospitals, or government agencies. Applications could, for example, bind external I/O devices and adjust the meeting room's lighting accordingly, before the user actually enters the room.

However, the concept of proactive adaptation is very challenging. The adapting entity has to be aware of upcoming context changes, i.e., be able to predict various context time series, as well as determine possible adaptations to cope with these changes. As adaptations are not limited to the adapting entity itself, it needs to be aware of available actuators and resources, including their respective capabilities. In order to optimize adaptation decisions, the set of possible adaptation alternatives has to be rated. As predictions are fluid and change frequently, this process has to be repeated over and over again. Additionally, adaptation in multi-user environments requires coordination in order to avoid oscillating effects.

Without suitable system support, the applications themselves are the adapting entities, leaving the challenges above to the application developers. This thesis presents a general framework, including middleware-based system support, for coordinated proactive adaptation in multi-user pervasive systems that eases application development by providing uniform access to context predictions and services, as well as calculating, rating, coordinating, and instructing adaptations for the applications.

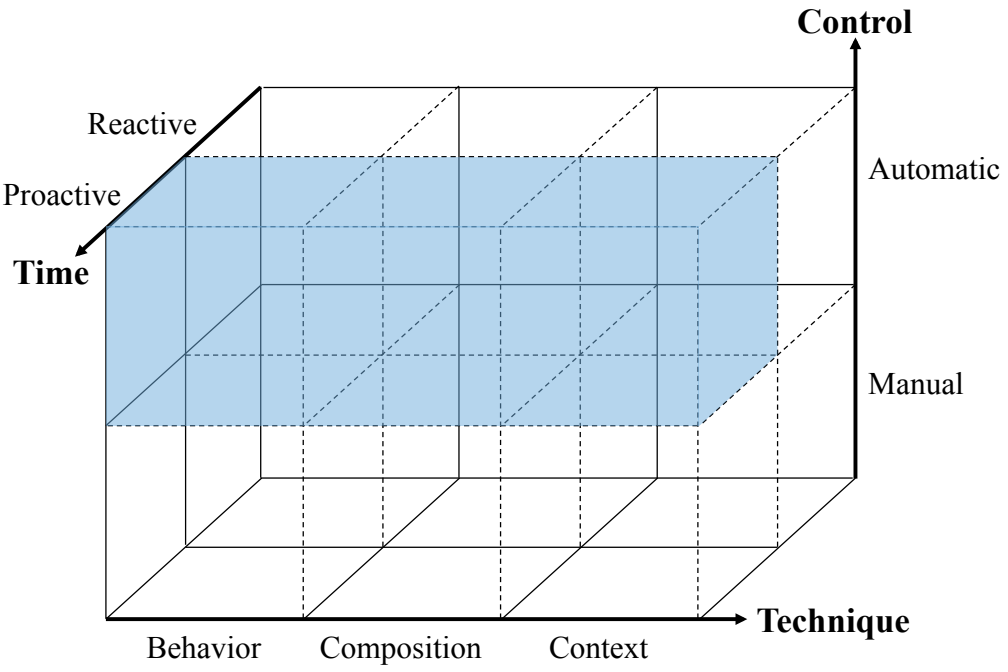


Figure 1.1: The topic of this thesis in the context of the dimensions of application-level adaptation in pervasive systems.

1.2 Research Questions

In a nutshell, applications in pervasive systems can adapt their behavior, their composition, as well as their context. Hereby, the adaptation control lies on a spectrum between manual, i.e., initiated by the user, and automatic adaptation, i.e., executed by the underlying system. Finally, adaptations can be proactive or reactive, i.e., before or after they become necessary. The goal of this thesis is to provide system support for automatic proactive adaptation in multi-user pervasive environments. Figure 1.1 visualizes the thesis' topic with regard to the different dimensions of application-level adaptation described above. More specifically, the thesis will answer the following research questions:

1. How can context sensing, predicting, and actuating services be accessed in a uniform fashion to support proactive adaptation?
2. How can an upcoming adaptation/adaptation sequence of an application be pre-calculated and optimized?
3. How can pre-calculated adaptation alternatives be coordinated to avoid interferences and oscillating effects while preserving optimization?
4. How can proactive and fallback reactive adaptation be combined in one automatic control loop?
5. How beneficial is proactive adaptation at what cost?

1.3 Contributions

This thesis presents a general framework and middleware-based system support for proactive adaptation in multi-user pervasive systems. The main contributions contained in this thesis are as follows:

First, a context interaction model including a set of service-transparent context queries based on the abstraction of context variables is developed in Section 5.2. The model supports proactive adaptation through its uniform access to sensing, predicting, and actuating services in the environment.

Second, an application configuration model with time-dependent utility and cost metrics is introduced in Section 5.3. The model allows to specify an application's context dependencies and optimize adaptation decisions with regard to the duration the respective application configuration is instantiated.

Third, a set of configuration and coordination algorithms for automatic adaptation are developed in Sections 5.4 and 5.5, respectively. The configuration algorithms search for all possible adaptation alternatives of an application based on current or predicted context. The coordination algorithms find interference-free adaptation plans for multiple applications in a shared context space, while optimizing the overall system utility.

Forth, a control loop is designed in Section 5.6 that combines automatic adaptation control for proactive and fallback reactive adaptation.

Finally, a prototype system is evaluated in extensive simulations of an interactive environment using real user traces in Chapter 7. The evaluation shows the benefits and costs of the framework, as well as proactive adaptation in general.

1.4 Structure

The remainder of this thesis is structured as follows. Chapter 2 introduces the theoretical background of the thesis. Afterwards, Chapter 3 reviews related work. Chapter 4 discusses the system model of the present thesis, as well as the adaptation support and the requirements of the framework for proactive adaptation. Based on this foundation, Chapter 5 presents the framework and system support for proactive adaptation in pervasive systems. Chapter 6 details the prototype implementation of the framework, before Chapter 7 evaluates it. Finally, Chapter 8 closes the thesis with a conclusion as well as an outlook on future research challenges.

2 Background

The last chapter motivated the present thesis, specified its research questions, and listed its contributions. This chapter introduces the theoretical background of the thesis, addressing the concepts of pervasive computing in Section 2.1, context-aware computing in Section 2.2, adaptation in pervasive systems in Section 2.3, proactive computing in Section 2.4, and context prediction in Section 2.5. Afterwards, Chapter 3 discusses related work.

2.1 Pervasive Computing

The terms *pervasive* or *ubiquitous computing* describe the paradigm shift away from traditional desktop computing to the current stage of the modern computing landscape, in which connected computational devices become interwoven with artifacts in our everyday life. Such IT-augmented environments were first described by Mark Weiser in 1991 [120]. Weiser argued that every prevailing technology evolves over time and, in that process, eventually reaches an ubiquitous state. Even earlier in 1978, Jef Raskin described the concept of *information appliances* in an internal document at Apple (according to [13] and [69]), anticipating a similar development away from all-purpose computers to task-specific devices, e.g., portable media players, digital cameras, and smartphones.

Already, the computer literate generation naturally uses information technology, adapting quickly and readily to innovations [49]. However, pervasive computing is not limited to human-computer interaction (HCI). Beyond fundamentally changing the way humans and computers interact, essentially by reducing the conscious interaction to a minimum, pervasive computing aims at the smart integration of independent computing devices. For this, processing, sensing, activation, and communication is embedded into devices and environments that interact with each other in order to provide a higher level of service to their users, forming smart environments. In such smart environments, devices detect

and analyze their physical surroundings, and applications adapt themselves as well as the physical world automatically. Finally, the users control these systems implicitly without even thinking about using a computer.

Next, the concept of context-aware computing is introduced, including a brief overview of the field's evolution.

2.2 Context-aware Computing

The concept of *context*, and therefore also its definition, has evolved steadily since the field of *context-aware computing* has emerged. In 1994, Schilit *et al.* [104] defined context in a top-down manner by formulating questions concerning the information that is vital for context-aware computing. They identified the user's location, the user's social group at the same location, and the nearby resources as the three key factors. Then, context-aware software adapts according to changes to this information. In the years following this initial definition, similar example-based definitions were given, all primarily focusing on the user's location, environment, identity, and times, e.g., in [18], [95], and [106]. Within the CyberDesk project [31], Dey extended the list even further by adding the user's emotional state, focus of attention, orientation, as well as the time of day. The set of identified context variables can be categorized as follows:

- *Physical environment*, such as location, time, and lighting,
- *Individual-related information*, such as the user's attention and social status at a specific location, and
- *Available computing resources and services*, such as network access-points, projectors, and printers.

However, definitions based on examples have the problem that newly identified variables, which influence the user's/application's context and, therefore, are relevant for context-aware software, may not fit any of the categories. In 1999, Dey and Abowd formulated a general definition, which focuses on HCI:

> ***Context*** *is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.* [32]

In analogy to the variety of definitions given for context, there are several different points of view about what context-awareness actually means. In 1995, as one of the first to address this topic, Schilit [105] described context-aware computing as the application's ability to detect context changes and react to them. In contrast, Dey [31] describes a context-aware user interface, leaving the adaptation decision to the human. More specifically, the system provides information and services according to the context and the user's task, but the adaptation decision itself is made by the user. Other projects focus on the system's flexibility [97], it's behavior [119], information selection [95] and tagging [79], as well as automated actions [17], to name a few. The range of context-aware applications can be categorized as follows:

- *Presentation* of information and services to a user,
- *Selection* of information and services,
- *Tagging* of context to information for later retrieval, and
- *Automatic execution* of a service for a user.

From the various research directions in the late 90's, Dey and Abowd again derive a general definition:

> *A system is* ***context-aware*** *if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.* [32]

This definition of context-awareness is very inclusive regarding the specific application of context information. Further, it purposefully leaves out how context information is actually obtained, allowing a distribution of functionalities between context-aware applications and their underlying system (cf. *context models* and *management systems* in Section 3.3). More comprehensive surveys on context-aware computing can, for example, be found in [10], [23], and [53].

Next, adaptation in pervasive systems is discussed in more detail, before Section 2.4 compares the concepts of proactive adaptation and proactive computing.

2.3 Adaptation in Pervasive Systems

In pervasive systems, mobile applications are able to use resources beyond the boundary of the devices they are running on. For example, they can make use of

I/O services in their environment, such as speech input and video output, as well as control physical conditions in the space using temperature or lighting regulation services. This capability of mobile applications to dynamically incorporate services into their runtime environment, as well as change their own behavior based on their context, is referred to as *application adaptation*. (The pervasive system itself can adapt on the *network* and *system level* [46]. Typically, such network and system level adaptations are reactions to broken links, high latencies, etc., and are not directly related to the user. Both network and system level adaptations are not in the scope of this thesis.)

The above mentioned examples of making use of services in the environment, context-aware changes in an application's behavior, and regulating physical conditions are different types of adaptation with regard to their *technique*. Hence, adaptation technique distinguishes between the adaptation of a component's *behavior*, its *composition*, and the adaptation of *context* itself [46]. Behavior adaptations target the parameters of the application, e.g., output volume or user notification styles, whereas composition adaptations target the application's structure, e.g., substituting I/O components or migrating application parts. Context adaptations, in contrast, target the environment outside the boundaries of the application using suitable actuators.

Adaptation control ranges from *manual* adaptation by the user to *automatic* adaptation by the underlying system. Satyanarayanan [101] refers to this as the application-awareness of adaptation, with the *laissez-faire* strategy representing adaptation on the application side, and the *application-transparent* strategy representing adaptation by the system. The laissez-faire adaptation strategy leaves the responsibility of adaptation to the individual applications. On the one hand, this strategy allows a great deal of flexibility, as even very application-specific functionalities and structures can be adapted. On the other hand, it increases the complexity of the application development process and, hence, that of the overall system. All adaptations, as well as their respective effects, have to be considered by all applications in the system, and implemented by the developers themselves. Examples for the laissez-faire approach are Speakeasy [34] and Prism [112]. Both approaches offer means for adaptation, but see the responsibility in the hands of the applications, i.e., the application developers. The application-transparent adaptation scheme takes the responsibility of adaptation from the applications by

offering a suitable runtime environment. For this, the application and adaptation logic are separated, and the system allows adapting the applications via suitable application architectures and infrastructure services. This allows adapting both the composition of the application, i.e., from which services on which devices it is composed of, as well as the behavior of the application, e.g., replacing a visual with a text-to-speech output. Examples for the application-transparent approach are PCOM [9] and Gaia [92]. Both use component-based application models that allow to adapt the structure of an application by (ex)changing their respective components in use. In case a bound component becomes unavailable, the systems automatically replace the missing component with one or a combination of several other components, respectively, that offer an equally suitable service. Other adaptive systems feature both laissez-faire and application-transparent aspects. For example, the middleware systems MobiPADS [21] and SOCAM [44] combine the concept of self-responsible individual application adaptation with system-wide adaptation in the responsibility of the middleware using defined profiles.

However, further research in the field has shown that a more specific classification of laissez-faire adaptation is needed. Manual adaptation requires interaction from the user, whereas adaptations that an application automatically instructs – e.g., based on user preferences – are human supervised. In [9], Becker *et al.* address the need and describe three levels of adaptation support:

1. *Manual adaptation:* The application presents adaptation possibilities to the user, who makes the decision.
2. *Application-specific automatic adaptation:* Each application has its own automated adaptation routines, which were implemented by the programmer. Although this approach increases user experience, predefined and, therefore, inflexible routines bear issues in changing environments.
3. *Generic automatic adaptation:* Application programmers define a set of required services their application needs for execution. The service provision, however, is in the responsibility of the underlying system, e.g., the operating system or middleware. Additionally, the user may be able to list preferences, which influence the decision in case of multiple options.

Finally, adaptations can be *reactive* or *proactive* with regard to the *adaptation time* [46]. That is, an adaptation can either be determined and executed as a reaction to changes in the application's context that prevent the application from

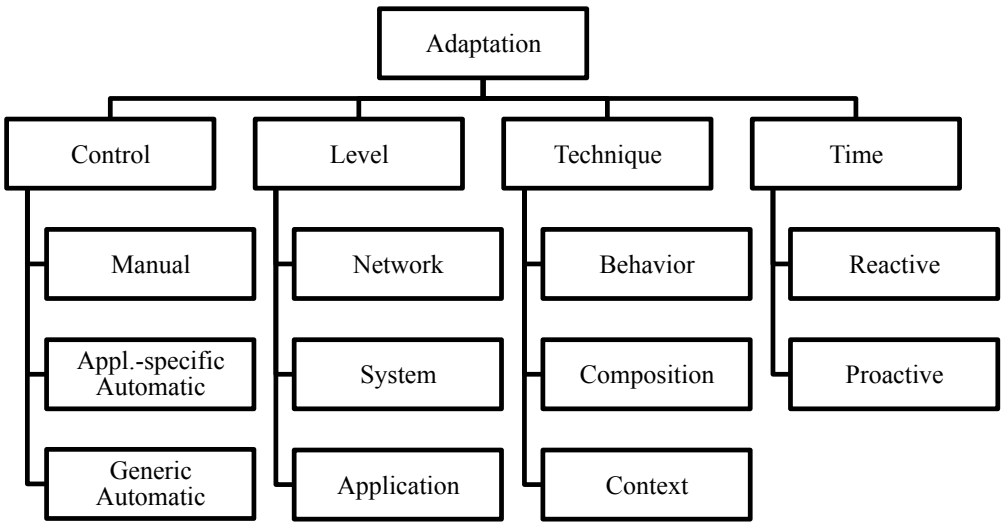


Figure 2.1: Categorization of Adaptation in Pervasive Systems

continuing its execution, i.e., *after* the event, or in anticipation of such changes, i.e., *before* an application can no longer be executed.

In summary, adaptation in pervasive systems is characterized by the four dimensions *control*, *level*, *technique*, and *time*. Figure 2.1 gives an overview of the individual adaptation categories grouped by these four dimensions. Further discussions on adaptation can, for example, be found in [7], [23], [46], and [61].

The next section reviews the differences between proactive adaptation and proactive computing, before Section 2.5 introduces context prediction.

2.4 Proactive Adaptation vs. Proactive Computing

In the context of pervasive computing, the term *proactive computing* was first introduced by Tennenhouse [114] in 2000. It is used to describe the evolution away from *interactive computing*, i.e., from classical human-centered workstation settings to human-(un)supervised pervasive computing scenarios. In [118], Want *et al.* further discuss proactive computing as well as the differences to *autonomic computing*. The aim of proactive computing are unobtrusive systems that connect to the physical world and require as little human interaction as possible. Further, they should anticipate the user's needs and act on his/her behalf. For this, Want *et al.* identify seven principles as foundations of proactive systems,

namely connecting with the physical world, deep networking, macro-processing, dealing with uncertainty, anticipation, closing the control loop, and making systems personal. Despite leading to similar techniques, autonomic computing, in contrast, describes the discipline of managing the complexity of a heterogeneous system through appropriate system design principles.

In [98], Salovaara *et al.* present their view of the concept of proactive computing. They suggest that a system can act proactively, if it has a hypothesis about what its user's goals are. In order to achieve these goals, the system makes use of different resources. The authors present a classification of six different types of proactive resource management in order to become a proactive system, namely preparation, optimization, advising, manipulation, inhibition, and finalization of user's resources.

Proactivity from an application adaptation perspective, on the other hand, is defined as *modifications of an application performed before an application can no longer be executed* [46]. Although this description of *proactive adaptation* interleaves with Tennenhouse's proactive computing, it is not congruent. As an example, in [115], the system automatically adjusts the lighting of the environment based on what it anticipates the user's desire is. The system connects to the physical world and acts on the user's behalf using anticipation, dealing with the uncertainty that comes with any type of reasoning. Hence, it is a classical example of proactive computing. However, it does so after it notices a change in the environment, i.e., in a *reactive* manner from an adaptation standpoint. In the adaptation terminology, acting on the user's behalf is a form of automatic adaptation, either application-specific automatic or generic automatic. Hence, in pervasive computing research, the term *proactive* can either refer to *before the user acts*, or *before the triggering event happens*, respectively. The main difference is that in order to act before an event takes place, the system must have knowledge of that event, which requires *context prediction* in addition to reasoning about the user's intent. Although small, this differentiation becomes rather important when discussing and classifying related research, as Chapter 3 will do. For instance, going back to the example above, [115] is not proactive in the terms of this thesis, but reactive with application-specific automatic adaptation control.

The next section discusses context prediction as a specific form of reasoning that aims at continuing context time series as accurately as possible.

2.5 Context Prediction

Proactive adaptation is based on predicted context information. The adapting entities prepare themselves for future context events, e.g., by pre-computing alternative runtime configurations. This includes forecasting the state of the pervasive environment as well as the availability of context services. The problem of *context prediction* is not one of the research questions of this thesis. Instead, as described in Section 6.4, the framework implements a set of popular approaches from which one is selected at runtime based on the specific prediction task.

There are two major categories within the overall task of predicting context for proactive adaptation. The first category is anticipating the future state of a context variable. This includes the anticipation of when a context variable will reach a certain state and how long it will remain in that state. Prominent techniques used in context prediction are, for example, probability models and pattern matching algorithms. [12] uses variable order Markov models to predict the most probably context state transitions. Petzold's state predictor [83] constructs a state machine per context variable that should be predicted. In his alignment approach [110], Sigg utilizes pattern matching techniques originating in computational biology to find the recorded context sequence most similar to the observed context sequence. Similarly, support vector machines [19] calculate the distance between the vectors of the observed and the recorded time series. Other approaches utilize self-organizing maps [59], such as in [111], and autoregressive-moving average models [121], such as in [55]. In general, the future sequence of context states, including the duration that a variable remains in a state, is projected forward by analyzing the history of the context variable.

The second category is forecasting the movement of users and objects, especially objects that provide sensor and actuator services. Predicting the movement of a single object is less challenging. In tracking systems, for example, location predictions are calculated based on movement trajectories using dead reckoning [122, 62]. Based on the movement vector of an object, i.e., its direction and speed, the algorithm provides an estimated pattern on which the object will move. In [76], the authors extend fingerprinting – a positioning approach using the signals of WiFi access points – by particle filter models in order to calculate trajectories. In contrast, anticipating the movement of object groups is far more

complex. In the group mobility model approach [116], single objects are clustered into mobility groups by recognizing mobility patterns and object relationships.

Finally, there are several different context prediction approaches regarding user presence, interest, and availability that use various data as input, such as calendar information, booking systems, weather prediction sources, and social networks [54]. Comprehensive surveys on state-of-the-art context prediction can, for example, be found in [16] and [109].

This chapter discussed the theoretical background of this thesis regarding the fields of pervasive, context-aware, and proactive computing, as well as adaptation in pervasive systems and context prediction. The next chapter builds on this background while examining related work on context-aware applications and adaptive architectures, as well as the more specific fields of context modeling and management, application configuration modeling, and adaptation control theory.

3 Related Work

The previous chapter reviewed the theoretical background of this thesis. This chapter examines related work. First, context-aware systems and adaptive architectures are discussed and categorized with regard to the analysis of the respective disciplines in Chapter 2. Afterwards, individual approaches to context modeling and management, application modeling, and adaptation control are discussed, before Chapter 4 presents the system model of this thesis, and derives the requirements towards system support for proactive adaptation.

3.1 Context-aware Systems

Context-aware applications and systems have been subject to research for more than two decades, with the context-aware actions ranging from presentation of information to automatic execution of services. Section 3.1.1 discusses context-aware presentation and selection, before Section 3.1.2 reviews research on context-aware tagging and execution.

3.1.1 Presentation and Selection

In recent years, there has been an increasing number of applications in the field of so-called *proactive services*, e.g., [52, 85, 22, 80]. Here, services are selected based on the current context of the user as well as his/her preferences. The user preferences are, for the most part, not explicitly specified by the users themselves, but rather reasoned about based on previous user behavior, which achieves a higher level of unobtrusiveness. Subsequently, the list of selected services are *proactively suggested* to the user, i.e., without the user having to request them. As discussed in Section 2.4, this type of proactiveness is, from an adaptation standpoint, an automatic action.

In Proactive Sensor Networks (PSNs) [2], sensors and actuators form so called context overlays in order to make distributed decisions, i.e., without a centralized entity, such as a context management, in the system. The network is proactive in that the sensors themselves – by themselves or in collaboration – pre-process their sensed data to the needs of the respective actuators. Such collaboration of sensors, e.g., in order to infer high from low level context, is referred to as *distributed context decisions*. PSNs are not a traditional context-aware application, as the users in this work are actuators rather than humans. SOCAM [44], on the other hand, automatically adapts context information based on rules depending on the specific domains an application goes through. That is, it transforms the information between different domains of the context ontology, as well as their level of abstraction.

The first work on proactive service selection was presented by Pitkäranta *et al.* in 2005 in the DYNAMOS project [85]. The system matches services based on their description to the context of its users, and proactively notifies them about critical situations of predefined interests. In [91], the authors report on their experiences of applying DYNAMOS for boaters in a marine community. Examples for selected information are discounts on gasoline or location-aware wind warnings. In [80], Pawar *et al.* present a geographically distributed patient monitoring system that utilizes proactive service selection techniques for their context-aware emergency response service. The approach is limited to the response service, which makes it application-specific.

In [52], Hong *et al.* present an agent-based framework, that aims at providing personalized services to its users based on their respective context history. It is proactive in the sense that it does this automatically using context reasoning, instead of being based on manually defined preferences. Therefore, the main challenge in this work is to infer the user preferences for the current situation from past situations, and then find a matching service. Similarly, ProWMS [22] is a context-aware web service pre-fetching strategy for mobile devices based on user preferences. The preferences are automatically computed from the service request history considering the current context.

In [88], Rasch *et al.* present an approach to proactive service selection specifically for pervasive environments. Central to the work is their formal context model called *Hyperspace Analogue to Context*. They model the capabilities of

services as well as user preferences as multidimensional geometric structures. The relation between these structures is used to calculate a service to preference matching metric. Finally, the system presents a ranked list of services to the user. Fenza *et al.* [35] address proactive service discovery as well, focusing on the healthcare domain. This work aims at providing personalized services depending on the patient's state of health, which is acquired from a network of wearable sensors. The two main technologies are Semantic Web for service and context modeling – specifically the Web Ontology Language (OWL) extension OWL-S – and fuzzy logic for context reasoning. Again, the system's output is a ranked list of services that match the patient's health context.

3.1.2 Tagging and Execution

The Stick-e Note system [79] – inspired by Post-It notes – allows users to attach digital information to physical objects, creating so-called *situated information spaces*. In similar fashion, the Augment-able Reality system [89] enables its users not only to consume augmented information, but also dynamically attach virtual information to real objects. Further, this augmented information is accessible via standard desktop PCs as well as wearable computers, which notify their users in case of new information in their environment.

In [28], Cooperstock *et al.* describe the evolution of a standard conference room into a smart, reactive environment in three iterations. The room supports its users by detecting intentions via triggering events and offering task-based support. Similarly, the smart meeting room system EasyMeeting [25] provides six context-aware services in support of both the speaker and the audience of a presentation, namely speech recognition, presentation, lighting control, music, greeting, and the so-called *profile display service* that shows the audience personalized information. In both systems, the execution of a service is triggered manually. In contrast, MobiPADS [21] offers automatic execution of services. For this, developers specify sets of rules in form of profiles, which are then used for triggering services in case predefined events were detected.

In the MavHome project [27], Cook *et al.* explore an agent-based smart home management system that acts on the user's behalf or supports the user in his/her task. The agents have a set of defined actions that are triggered based on the

Project	Action				Control	
	P	S	T	E	M	A
CyberDesk [31]	+				+	
Cyberguide [1]	+				+	(+)
GUIDE [30, 26]	+				+	(+)
SOCAM [44]	+					+
PSNs [2]	+					+
DYNAMOS [85]		+				+
Pawar *et al.* [80]		+				+
Hong *et al.* [52]		+				+
ProWMS [22]		+				+
Rasch *et al.* [88]		+				+
Fenza *et al.* [35]		+				+
Stick-e Note [79]			+		+	(+)
Augment-able Reality [89]			+		+	(+)
Cooperstock *et al.* [28]				+	+	
EasyMeeting [25]				+	+	
MobiPADS [21]				+		+
MavHome [27]		(+)		+		+

Action: P = Presentation, S = Selection, T = Tagging, E = Execution
Control: M = Manual, A = Automatic
+ = Explicit feature, (+) = Implicit feature

Table 3.1: Overview and Classification of Context-aware Systems

predicted next user task/goal, such as, for example, starting the lawn sprinklers or placing a food order. For this, it combines the strength of several prediction approaches into the meta predictor *Predict*^{2}. MavHome offers implicit context adaptation via actuators, such as automated blinds, but is an action-based framework and not an adaptive architecture. That is, the system does not feature a control loop that specifically monitors and adapts a certain context, but aims at automating the actions that the inhabitant would instead have to carry out.

3.1.3 Overview and Classification

Table 3.1 shows a classification of context-aware systems with regard to their context-aware action (*presentation*, *selection*, *tagging*, and *execution*), as well as control (*manual* and *automatic*). The systems for automatic service selection

typically apply learning-based algorithms to reason about the user's preferences, utilize this information to filter out unwanted services, and let the user choose which service should be employed. The systems offering service execution, on the other hand, typically apply pre-defined rules for offering actions to their users, or automatically triggering actions on behalf of their users, respectively. MavHome additionally incorporates context prediction to reason about the user's next task.

The next section reviews adaptive architectures with the focus on application-level adaptation.

3.2 Adaptive Architectures

Various adaptive frameworks have been developed over the past decade. They enable mobile applications to adapt their behavior, incorporate remote resources and services, as well as actuate context at runtime. Next, Section 3.2.1 discusses reactive adaptation architectures, before Section 3.2.2 reviews proactive adaptation architectures.

3.2.1 Reactive Architectures

There exists a variety of research on adaptation frameworks for pervasive environments. For reactive adaptation, there are, for example, 3PC [46], Aura [39], and Gaia [92]. All three provide support for behavior and composition adaptations, even beyond the application level. They each feature a layered architecture made up of single components with different adaptation responsibilities.

In the 3PC project, the middleware BASE [11] – designed for so-called *smart peer groups* that do not rely on infrastructure – automatically switches between network plug-ins depending on the state of the respective link. SANDMAN [103] adapts the system by forming service clusters based on group mobility, in which redundant services are set to sleep in order to achieve higher energy-efficiency. The PCOM layer [9] offers generic automatic adaptation of its component-based applications using contracts. Finally, COMITY [63] coordinates individual application adaptations in case of context interferences. (As the basis for adaptation coordination in this thesis, COMITY is discussed in more detail in Section 5.5.1.)

Similarly, Aura's Odyssey [100] and Coda [102] offer adaptation at the network and system level, respectively. On the application level, Spectra [37] automatically distributes functionalities amongst entities in light of the resource restrictions of mobile devices. Finally, the task manager Prism [112] adapts the application's composition proactively using task to tool mappings. Prism is discussed in more detail in Section 3.2.2.

Within Gaia, the reflective object request broker dynamicTAO [93] adapts the communication protocols and policies that are used by applications according to changes in the environment. Above, the Model-Presentation-Controller-Coordinator (MPCC) application framework [48] decouples the individual application components for manual adaptation, e.g., substitution of services. Finally, Olympus [87] complements MPCC to offer automatic application composition adaptation based on predefined functionality to service mappings.

Many other reactive systems exist that adapt automatically on different levels using different techniques. With Speakeasy [34], users can manually adapt the composition of their applications. The RUNES middleware [29] offers dynamic reconfiguration of components, i.e., adaptation on the system level, as well as the composition of components that form an application, i.e., adaptation on the application level. Similarly, P2Pcomp [36] and REFLECT [108] create adaptable applications by using late binding of components, i.e., at runtime, instead of at compile time using predefined component dependencies. Additionally, applications in the REFLECT system are able to adapt their behavior individually.

One.world [43, 42] and O2S [78] focus on the development of adaptive applications. More specifically, one.world provides a comprehensive runtime architecture that aims at simplifying the development process, e.g., by offering data management and event processing. In O2S, on the other hand, applications are *goal*-oriented and adapt their behavior by substituting the *technique(s)* they use – not to be confused with the *adaptation techniques* – to reach their goals.

Similar to PCOM, iROS [56] supports automatic adaptation of the composition of its loosely coupled applications. That is, iROS' event heap selects appropriate components from its repository based on their descriptions. In case of errors, for example crash of a component, the system resends events to an alternative component with the same capabilities, if available.

In [20], Byun *et al.* present an approach that learns user preferences for human supervised environment control from context history. The system is application-specific automatic instead of generic automatic, as it is not a framework for applications, but monitoring and adapting the context is the application itself. The authors further discuss different uncertainty issues in general with regard to control and adaptation, and propose possible solutions for these problems. Similarly, Vainio *et al.* [115] present automatic context adaptation based on fuzzy logic techniques for smart home environments. That is, the system monitors the state of an adaptable context, such as lighting and temperature, learns the user's routines, and subsequently adapts the context under the user's supervision. Due to the use of fuzzy rules, the system can control the environment even in situations of uncertainty, i.e., ones it has not yet learned. As in the work by Byun *et al.*, controlling the environment is the application itself.

Adaptation in the above systems is reactive, i.e., they determine and execute the adaptations after the triggering event. As an exception, Aura's task manager Prism anticipates adaptations for some aspects of the system, such as network load and data distribution. Prism and proactive frameworks are discussed next.

3.2.2 Proactive Architectures

Prism [112] – the task manager in the Aura [39] architecture – prepares adaptations on the application level that are related to the next anticipated user tasks. For example, the system uses approximate location predictions based on location data in calendar entries in order to prepare data transfer to a specific location. For this, Prism uses a set of tasks to application mappings, e.g., *edit text* to Microsoft Word, that are provided by the system administrator. The adaptation itself is then triggered by the user.

Adaptable Pervasive Flows (APFs) [47] are workflow-like models of an entity's activity that adapt with regard to that entity's situation. The flows consist of a series of tasks – either representing atomic services or a *subflow* – that are connected by so-called *context-aware transitions*, i.e., transitions defined by context events, such as a location change. Adaptation in the APF system is generic-automatic. Developers define a set of goals and constraints for each flow, and the *flow system* calculates the specific adaptations of a flow's composition. These

Project	Technique			Control			Time	
	B	COM	CTX	M	A	G	P	R
Mayrhofer [66]	+				+		+	
Speakeasy [34]		+		+				+
MPCC (Gaia) [48]		+		+				+
Prism (Aura) [112]		+		+			+	
RUNES [29]		+			+			+
O2S [78]		+			+			+
one.world [43, 42]	+	+			+			+
REFLECT [108]	+	+			+	+		+
Spectra (Aura) [37]		+				+		+
Olympus (Gaia) [87]		+				+		+
iROS [56]		+				+		+
PCOM (3PC) [9]		+				+		+
P2Pcomp [36]		+				+		+
APFs [47]		+				+	+	
Byun *et al.* [20]			+		+			+
Vainio *et al.* [115]			+		+			+
CALCHAS [15]			+		+		+	
COMITY (3PC) [63]	(+)	(+)	+		+			+

Technique: B = Behavior, COM = Composition, CTX = Context
Control: M = Manual, A = Appl.-specific automatic, G = Generic automatic
Time: P = Proactive, R = Reactive
+ = Explicit feature, (+) = Implicit feature

Table 3.2: Overview and Classification of Application Adaptation Architectures

are either *horizontal* adaptations of the flow in case re-planning is necessary, or *vertical* adaptations in case a task is substituted by a subflow or its mapping to an atomic service became invalid. The APF system uses context prediction in order to anticipate these adaptations.

In his dissertation [66], Mayrhofer presents a general architecture for context prediction that allows applications to query future context information in order to proactively assist its users. The focus of the work is a five step prediction process: (1) gather context information from a heterogeneous network of sensors, (2) extract features, (3) classify, (4) label, and (5) predict. As the framework specifically offers an interface for applications to access predictions, it is not only an approach to context prediction, but also an adaptive architecture.

In [15], Boytsov and Zaslavsky present the CALCHAS system, which offers context predictions to applications that, in turn, use actuators to adapt their context. In order to support proactive adaptation, they further present an extension to the context spaces theory [77] with the concept of context adaptation via actuators. The authors see proactive adaptation as reinforcement learning tasks that aim at improving both the predictions as well as the adaptation decisions. Actually, the focus of the work is on the quality of the predictions and decisions, and not on the framework for adaptation. The system follows the laissez-faire/ application-specific automatic approach to application adaptation, as it provides the predicted context information to the applications, but does not adapt the context itself.

3.2.3 Overview and Classification

Table 3.2 shows an overview and classification of adaptive architectures with regard to their adaptation technique (*behavior*, *composition*, and *context*), their adaptation control (*manual*, *application-specific automatic*, and *generic automatic*), and their adaptation time (*proactive* and *reactive*). The overview shows that the research community predominantly focuses on the compositional adaptation of applications, i.e., creating applications from individual, interacting components in the pervasive system, and adapting their structure by substituting individual services. Further, the classification indicates that proactive architectures are not yet fully researched.

Next, work in the fields of context modeling and management, application configuration modeling, and adaptation control theory is discussed.

3.3 Context Models and Management

Context models define how systems and applications interact with their physical environment, whereas *context management* describes the acquisition, processing, and distribution of context information. *Context management systems* handle heterogeneous context sources that differ in their data representation, interface, etc. Processing of the acquired data includes composing datasets from

single data, reasoning about higher context based on defined rules, e.g., recognizing situations from audio and location data as in [25], as well as distributing data throughout the environment. There are several different approaches and architectures depending on the type of context information and use case(s) of the system. In the following, a set of popular approaches are discussed as representatives of the field.

Nexus [51] is a platform for location-aware applications. The objective of the collaborative research center's project was to model the real world, plus additional virtual objects associated with locations, into so-called *augmented areas*, and provide this information to location-aware applications via a suitable architecture. Integrated prediction algorithms, especially dead reckoning, allow not only to query recorded information, but also request location predictions. However, Nexus does not offer any adaptation support beyond acquiring location data.

In Aura [39] – the middleware-based system that is designed to support its users in everyday tasks – Prism [112] manages the user's tasks using a task-based application model, and monitors the respective task-relevant context, e.g., location and available resources. That is, Prism's context observer component reports changes in the monitored context to the task and environment managers, and the system subsequently offers adaptation options to its users for manual selection according to the changes.

CoBrA [25] is a context reasoning and distribution approach for defined smart spaces, such as smart meeting rooms. The system uses an ontology-based context model based on SOUPA [24]. Using ontology-based reasoning, the centralized context broker provides predefined services to scenario-specific applications, such as the smart meeting room system EasyMeeting. Although CoBrA allows applications to automatically and actively influence their context, e.g., dimming the light or starting the presentation, these adaptations are limited to an application-specific and static set of services. Context prediction is not addressed.

SOCAM [44] is a middleware-based framework for context-aware applications that offers context information and reasoning as distributed services in dynamic environments. Central to SOCAM is its context interpreter that features ontology-based context reasoning. The CONON ontology [45] includes,

in contrast to SOUPA, information quality metrics that could be extended to support context prediction. However, in its current state, SOCAM does not offer context prediction. Further, context actuation is in the responsibility of the applications.

Even though context management systems are a well researched topic, they do not offer sufficient support for proactive adaptation. Most approaches do not address context adaptation via actuators. Further, traditional context management systems do not provide desirable support for context prediction, e.g., monitoring of context and notifications in case of updates. A more extensive discussion and categorization of approaches can, for example, be found in [5], [58], and [96].

3.4 Application Configuration Models

Application configuration models – or simply *application models* – are used to automatically calculate valid runtime configurations of applications based on descriptions of their respective requirements – a prerequisite of generic automatic adaptation. As with context models and management systems, the approaches vary depending on the system-specific objectives.

Olympus [87] is an abstract programming model for Gaia's active spaces. With it, the developer can specify requirements of an application without knowledge of how they are implemented. The system uses the modeled requirements to configure the active space using a utility model to select the best implementation. Mukhtar *et al.* [67, 68] present an approach for configuring multimedia services based on user preferences and device capabilities. In order to compare the devices' usability for a task, the user preferences are transformed into a tree structure, with the leaves representing the accumulated user rating. Similarly, in PCOM [9] and iROS [56], applications are composed of services based on service descriptions, and the systems automatically adapt the application's composition, if necessary, as discussed in Section 3.2.1. Finally, MADAM [40, 38] is a middleware specifically developed for mobile and adaptive applications in highly dynamic environments. In MADAM, applications are also component-based. The components that form an application are selected using a utility function following a maximum utility strategy.

Next to adapting ahead of time, proactive adaptation aims at optimizing adaptation decisions based on the future progression of the context time series, including the respective duration of a context. However, in the application models above, utility and costs over time are not considered. Hence, they are not fully suitable to support proactive adaptation.

3.5 Adaptation Control Theory

In adaptation control theory, there are three reference control cycles. First, there is the MAPE-K control loop [4, 57], as in the phases of *monitor*, *analyze*, *plan*, and *execute*, as well as the loop's underlying *knowledge* base. Second, the Autonomic Control Loop [33], consisting of the similar *collect*, *analyze*, *decide*, and *act*. Finally, the more generic Observer/Controller Architecture [90] that aims at highly complex systems with unexpected behavior.

Naturally, any approach to adaptation control features the same succession of stages. However, adaptation control for proactive adaptation (see Section 5.6) differs in three aspects. First, it suspends its cycle during the *plan*/*decide* stage until the triggering event happens. That is, appropriate adaptations to anticipated changes in the context are calculated and stored for later retrieval. However, the specific decision on which adaptation is to be executed, if multiple options exist, is made at the time of the actual change. Second, the first phases leading up to that suspension, i.e., *monitor*, *analyze*, and the calculating part of *plan*, form a smaller loop themselves. For each predicted context change, the system iterates through this smaller loop and stores the resulting adaptation plans. Finally, proactive adaptation control additionally requires a fallback loop in order to be able to adapt reactively. That is, in case there is no pre-calculated adaptation plan – whatever the reason – the system has to calculate and execute one immediately.

This chapter reviewed related work with regard to context-aware applications, adaptive architectures, context modeling and management, application modeling, and adaptation control. The next chapter presents the system model of the present thesis, discusses its adaptation support, and derives the requirements towards a framework for proactive adaptation.

4 Proactive Adaptation in Pervasive Systems

The last chapter discussed related work regarding context-aware applications and adaptive architectures. This chapter first describes the thesis' underlying system model as well as its level of adaptation support. Afterwards, it derives the requirements towards a general framework and system support for proactive adaptation, which is subsequently presented in Chapter 5.

4.1 System Model

In this thesis, a *pervasive system* consists of a set of users, devices, and *pervasive applications* that provide services to their users. These pervasive applications make use of the available resources, functionalities, and context services in the environment. They are context-aware and able to adapt to changes in their context. That is, the set of resources, functionalities, and context services that an application can respectively use to provide its service are referred to as a *functional configuration*. The applications are able to switch between their functional configurations, depending on how the resources, functionalities, and context services change. The active set of resources, functionalities, and context services is referred to as the application's *(functional) configuration instantiation*.

The applications in the system are able to specify their requirements towards their context – as further discussed in Section 5.3 – in order for a centralized entity to reason about all application requirements/functional configurations. The requirements describe the physical conditions at the location of the application, as well as required external services, based on context variables, such as `TEMPERATURE` or `VISUAL_OUTPUT`. Further, the pervasive applications are assumed to be *cooperative*. That is, applications are truthful about their requirements and their functional configurations' respective utility functions, as well as compliant with all adaptation instructions they receive from the centralized entity. The compliance aspect also holds true for actuators and any other services.

In order to participate in the system, all devices are equipped with appropriate system software, in this instance the middleware BASE [11]. BASE is a middleware that has been specifically designed for pervasive environments. It has a lightweight but extensible core, which enables its operation on resource-poor devices, such as embedded systems, but also supports costly functionalities running on full-fledged devices, such as desktop computers. Devices which are equipped with BASE are able to detect each other and form a spontaneous network. In order to build and execute pervasive applications, BASE models functionalities and device capabilities as services and provides a uniform access. Each (remote) service can be accessed via local proxies implementing a defined interface. Moreover, BASE enables remote communication while shielding applications from the underlying communication technologies, interoperability protocols, and communication models.

Further, the existence of a *location model* such as [99] or [6] is assumed. The location model provides symbolic location descriptions with room-level granularity, e.g., as provided by the Active Badge system [117], and allows for position, distance, range, and nearest neighbor queries [8]. Finally, this work assumes highly dynamic environments, in which any device and service may frequently leave and (re-)enter the network. BASE already provides dynamic environment support on the communication layer, but the framework needs to cope with the unavailability of services as well.

The next section discusses the level of adaptation support that should be provided by the framework with regard to the categorization of adaptation derived in Section 2.3.

4.2 Adaptation Support

In this thesis, adaptation is limited to the *application level*, i.e., *network* and *system level* adaptation are not addressed. For this, the pervasive applications instantiate different functional configurations. Switching between different functional configurations may require using several adaptation *techniques*. The adaptation can have an effect on how the application *behaves*, which external resources the application uses, i.e., its *composition*, as well as directly influence the physical environment via actuators, i.e., the application's *context*.

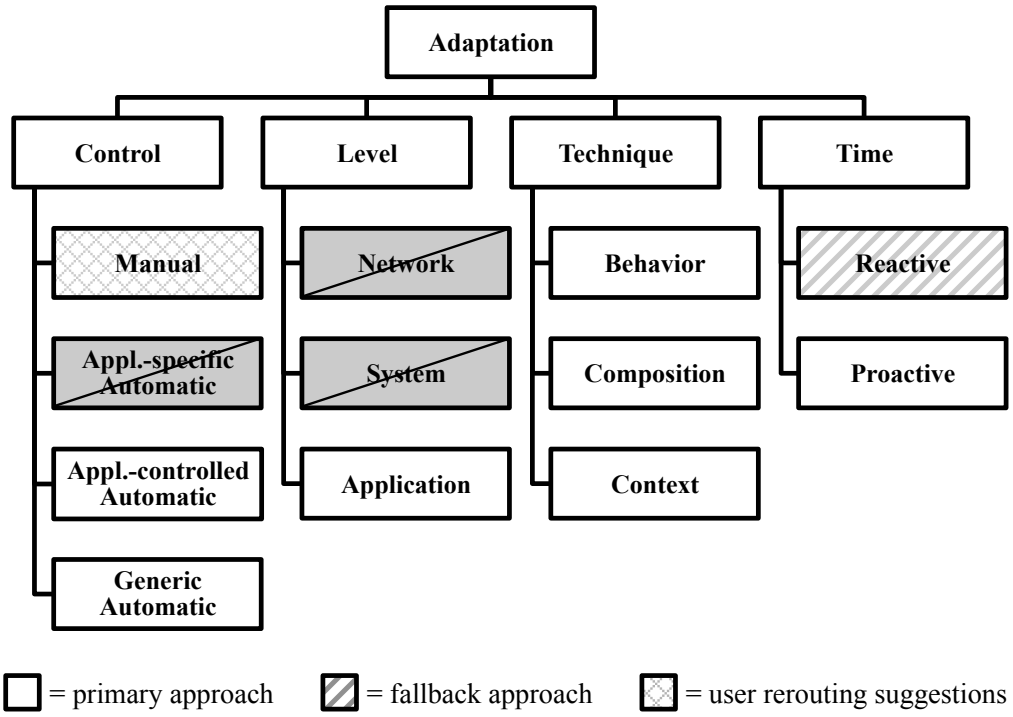


Figure 4.1: The adaptation support provided by this thesis with regard to the categorization of adaptation in pervasive systems.

Depending on the specific technique, different forms of adaptation *control* are used. Ideally, the adaptation framework should calculate and instruct adaptations based on the application's requirements (*generic automatic* adaptation). However, with the exception of context adaptation via actuators, the actual execution of a behavior or composition adaptation is in control of the application. Hence, this thesis introduces an additional control type called *application-controlled automatic* adaptation. It describes the distribution of responsibilities between the centralized management entity, which determines the adaptation plan, and the application that executes the adaptation instructions. User adaptations, i.e., rerouting him/her, are an exception when it comes to the adaptation control. It is reasonable to assume cooperative applications, but neither realistic nor desirable to assume cooperative humans. In this work, user adaptations are suggestions by the framework in case the requirements of the user's application can not be fulfilled. Hence, they can be regarded as *manual* adaptations.

Finally, proactive adaptation is defined as adapting *before* an application can no longer be executed. This not only requires knowledge of the changes that will take place, but also exactly when they will take place in the magnitude of milliseconds, in order to instruct the adaptation at that point in time. Unfortunately, such precision in context prediction is not feasible as soon as human behavior/decision making is involved. However, it is possible to prepare for the upcoming changes and trigger the adaptation as soon as the corresponding event takes place. Hence, in this thesis, adapting to a pre-calculated functional configuration, i.e., without the delay of determining the adaptation, is considered to be a proactive adaptation.

Figure 4.1 summarizes the adaptation support provided by the framework developed in this thesis with regard to the prioritization of the single categories. The primary approach is proactive application adaptation using any technique with adaptation control in the responsibility of the system as much as possible, including a mixture of adaptation techniques and controls per adaptation. As a fallback option, e.g., in case a context event was not predicted, the framework additionally provides reactive adaptation.

Next, the requirements of providing the described level of adaptation support are derived, before the framework is presented in the following chapter.

4.3 Requirements

As stated in the research questions in Section 1.2, the problem of providing system support for proactive adaptation in pervasive system can be categorized into three major components. First, a context model and management system must provide a suitable abstraction to context. Second, an application model must allow applications to specify their context and external resource dependencies, such that the framework can calculate the application's adaptation options. Third, an adaptation control component must be able to monitor the environment and trigger necessary adaptation procedures.

In the following, Section 4.3.1 discusses the requirements towards these three components in detail, before Section 4.3.2 presents the requirements towards the configuration and coordination algorithms.

4.3.1 Component Requirements

This section presents the requirements of system support for proactive adaptation towards the three central components of the framework.

4.3.1.1 Context Model and Management

A context management system's basic responsibility is to acquire information via the sensors deployed throughout the environment, process it to fit the abstract context model, and finally provide it to the context-aware consumers. However, several context management approaches have an increased functionality, e.g., they offer smart environment services. In this work, the system's responsibility is also enhanced to the extent that it manages all context interaction, in order to create a single point of access for all entities in the system. For this increased role, the management's interface, or query set, respectively, becomes a key component of the system. Extending the traditional set of requirements – context acquisition and provision, a suitable interface, and dynamic environment support – three additional requirements are identified that enable a context model and management component to support proactive adaptation.

1. **Automatic Update Notification:** Predictions by adaptive approaches – i.e. ones capable of learning – change with their respective data set, e.g., with newly learned information or changes to the pervasive environment. This has implications for requesting applications as well as chains of interdependent predictions. However, regular update requests by these components can lead to flooding. Hence, the system should offer means for automatic update notifications.
2. **Context Adaptation as a Service:** Proactive adaptation is not restricted to preparing for future context events, but also includes influencing or preventing them. To do so, applications use so-called *context actuators*, such as light switches, air conditioners, or computing resource coordinators. Comparable to the task of context provision, the system should offer context adaptation as a service.
3. **Uniform Abstraction:** In a proactive system, applications must not only be able to request current and reasoned-about context from the responsible

components, but also influence context via actuators. That is, interaction between applications and their context is now bidirectional. Hence, the abstract interaction model must support both directions – ideally in a uniform fashion.

4.3.1.2 Application Model

With generic or application-controlled automatic adaptation control, determining the adaptation of an application based on the application's requirements is in the responsibility of the framework. In general, *application models* allow applications to specify their runtime requirements. In an adaptive framework, these runtime requirements are **specifications of physical conditions** as well as **necessary resources and context services**.

Further, a major benefit of proactive adaptation is the possibility of optimizing adaptations and adaptation sequences. To do so, the different adaptation options need to be rated using **a comparable, duration-dependent metric**, such as goodness, reconfiguration costs, and user preferences, that reflects the benefits and costs of an application configuration instantiation depending on how long the context will be in the respective state. Hence, the application model in this thesis should provide means for applications developers and/or users to specify such parameters and functions.

4.3.1.3 Adaptation Control

Section 3.5 discussed adaptation control theory, and briefly described differences between the traditional reactive control loops and a proactive control loop. These differences emerge from the additional requirements of **managing pre-calculated adaptations** and providing **a fallback reactive adaptation loop**. That is, next to the reactive (fallback) loop – i.e., monitoring context, analyzing situations, planning adaptations, and executing adaptation instructions in one continuous process – an adaptation control component for proactive adaptation has to run through two smaller, discontinuous loops simultaneously. In the first, it has to monitor context predictions, pre-calculate and pre-coordinate adaptations, and store the results for later retrieval. In the second, it has to monitor context changes and trigger the pre-determined adaptations.

4.3.2 Algorithm Requirements

Adaptation in multi-user environments involves two search problems, both of which part of the system support framework. First, the framework has to calculate the adaptation alternatives per application given one context – current or future – and the respective application requirements towards its context, i.e., *determine* adaptations. Second, in the case that multiple users/applications share a context space, the framework has to find an assignment of adaptation alternatives to applications without conflicting instructions, i.e., *coordinate* adaptations. However, in contrast to isolated applications and reactive systems, it is not sufficient to find one solution per search problem, respectively. In a framework for proactive adaptation in multi-user systems, the search algorithms have the following requirements:

1. **Complete Search:** For both search problems, the algorithms should be guaranteed to find a solution if one exists.
2. **Exhaustive Search of Adaptation Alternatives:** The set of determined adaptation alternatives of all applications in the same situation are the input of the adaptation coordination search algorithm. In order to not forestall possible solutions during the coordination search, the determination search should output all solutions to its respective search problem, regardless of its utility or any other factor.
3. **Optimal Solution for Coordination Problem:** As mentioned, the optimization of adaptations is essential to proactive adaptation. Hence, the coordination algorithm should find the optimal solution to its search problem. However, as the optimal assignment of adaptation alternatives to applications would be chosen every time per distinct situation, the algorithm may terminate after finding the optimal solution.

This chapter presented the system model, featured adaptation support, and requirements of this thesis. The next chapter presents the theoretical framework and centralized system support for proactive application adaptation in the previously described pervasive system.

5 System Support for Proactive Adaptation

The preceding chapter discussed the system model of this thesis and derived the requirements of providing system support for proactive adaptation. This chapter subsequently presents a general framework for proactive adaptation, including system support in form of a generic middleware extension. First, Section 5.1 gives an overview of the various aspects of the framework, as well as their structuring in separate components. Afterwards, Section 5.2 describes the context interaction model, and presents the centralized context broker that mediates all context interaction, ensuring uniformity. Section 5.3 presents the configuration model for applications in the system, including how applications specify their context requirements, and how the framework rates adaptations based on cost and utility metrics. Based on these application requirements and the given context, Section 5.4 determines the adaptation alternatives of an application using constraint satisfaction modeling and backtracking-based algorithms, enabling generic automatic adaptation. In Section 5.5, the individual adaptation alternatives are then coordinated in order to avoid context interferences in the multi-user system. Finally, Section 5.6 presents the adaptation control loop that combines the individual components of the framework by monitoring context and context predictions based on the applications' requirements, triggering (pre-)calculation and (pre-)coordination of adaptations, and issuing adaptation instructions to the respective entities, before Section 5.7 closes the chapter with a summary.

5.1 Framework Overview

Proactive adaptation in pervasive environments requires several steps. First, the adapting entity needs to be aware of its current context, as well as upcoming changes. Second, the entity has to be able to determine necessary adaptations based on this information. That is, the entity needs a specification of its so-called *functional configurations*. Functional configurations are sets of environment con-

ditions, such as lighting levels or temperatures, and necessary remote services and resources, such as audio inputs or visual outputs, that define the behavior, composition, and context influences of the entity. Based on these functional configurations and the current/predicted context, the entity needs to calculate adaptations that lead to a valid *configuration instantiation* in the entity's current/future circumstances. Such adaptations can be limited to the entity itself, i.e., switching to a different functional configuration, and/or include adapting the entity's context via actuators. In case of context adaptations, the adapting entity additionally needs to be aware of the available actuators in the environments, as well as their respective capabilities. Third, the entity has to enforce the adaptations, e.g., change its own behavior and give instructions to actuators.

However, this already challenging process only describes a basic proactive adaptation scheme. Ideally, adaptations should be optimized for a series of upcoming context events in order to avoid frequent switches of configuration instantiations. As an example, the best decision might be to switch to a configuration instantiation that is not ideal for a short period, but is valid throughout a series of context changes. As a result, the entity should go through the process for all predicted events – immediate and beyond – as well as compute and rate all possible adaptation alternatives for each of them. This leads to a constant effort of monitoring predictions, finding all solutions, and deciding on a strategy. Further, adaptation in multi-user environments requires coordination in order to avoid oscillating adaptations caused by so-called *context interferences*. Such context interferences occur when two or more entities influence a shared context differently, without being aware of the fact. The need for coordination also holds true for proactive adaptation. In addition to alternately preventing the entities from providing their respective functionality, the oscillating effects further nullify any benefits from the proactive scheme, resulting in the waste of resources.

Handling all the required steps for proactive adaptation is not sensible for single applications running on the typical resource-restricted devices that make up pervasive environments. The complexity of the necessary algorithms, as well as communication and managerial overhead, result in high delays and short battery lives. The framework presented in this thesis offers middleware-based support for proactive adaptation via centralized services. Hereby, the framework builds on existing work on context-aware frameworks and context prediction.

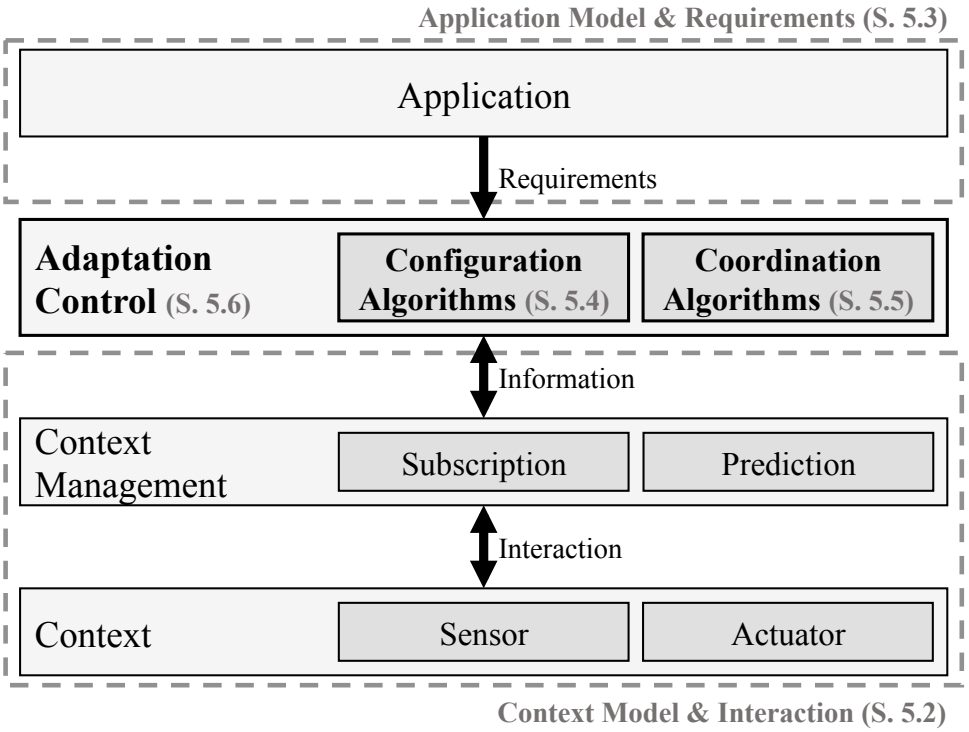


Figure 5.1: Framework Overview

Figure 5.1 shows a rough overview of the framework with references to the sections, in which the respective components/areas are presented in detail. The *context management* component (Section 5.2) provides context interaction – i.e., sensor and actuator services – as well as context prediction, in a uniform, distribution transparent fashion. That is, the component offers its services via a defined query language that uses a variable-based context abstraction. In case of prediction requests, the component selects the best-fit prediction approach based on the specific prediction task. For any of the information queries, consumers can subscribe to update notifications. In order to use the framework, *applications* have to provide their respective runtime model in the form of requirements and preferences (Section 5.3). Requirements specify the context dependencies of an application, e.g., certain physical conditions the application/user needs, or external services the application has to use in order to provide its function. The framework then uses its *configuration algorithms* (Section 5.4) to determine and rate the set of possible adaptation alternatives, each ensuring the execution of the respective application in isolation. However, pervasive environments are rarely populated by

one application alone. In shared environments, context dependencies from multiple applications can interfere with each other. Hence, adaptations in multi-user environments must be coordinated. That is, the framework uses its *coordination algorithms* (Section 5.5) to determine the set of adaptation alternatives – one per application – that maximizes the global utility in the environment. Finally, the *adaptation control* component (Section 5.6) implements the adaptation control loop. For all applications that register with their respective requirements, the component monitors relevant context information/predictions, triggers configuration and coordination algorithms, and instructs necessary adaptations to either the context management component in case of context adaptations, or the respective application in case of behavior or composition adaptations.

The following sections present the theoretical concepts behind interaction in the system and the various modelings, as well as the algorithms and processes used in the framework. Afterwards, Chapter 6 describes implementation details of the prototype system.

5.2 Context Interaction Model

Context is defined by its identity, location, and point in time – commonly referred to as the *primary context* [10]. In the well-researched reactive systems, identity and location are handled by context and location models, respectively [5, 8]. The point in time often is *right now* by default and not regarded in these systems. Further, the systems are only designed to provide context information to applications, not influence the context. In order to support proactive adaptation, it is necessary to be able to query context regarding other points in time, e.g., in the future while requesting predictions, as well as explicitly adapt the context, i.e., instructing changes instead of querying states.

This section presents a uniform context interaction model – i.e., accessing (distributed) sensing, predicting, and actuating services – that is suitable for proactive adaptation. For this, the abstraction of *context variables* as an extension of location models are introduced. Then, all context services in the environment are linked to their respective variable and location using service descriptions, and a set of service-transparent *context queries* are used for context interaction via a centralized *context broker*.

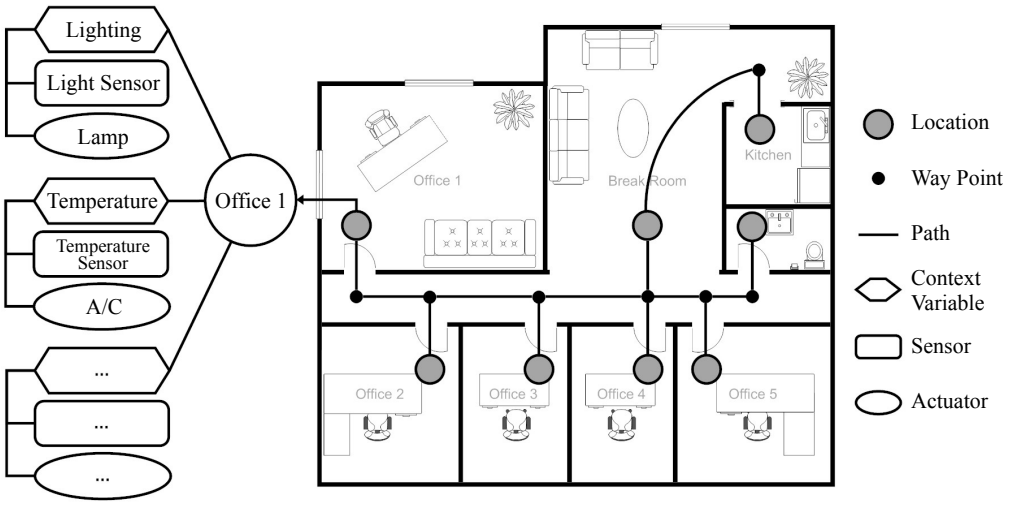


Figure 5.2: Variable-based Context Interaction.

5.2.1 Variable-based Abstraction

In order to abstract context into a representation that can be stored and processed, each context object must be modeled. That is, every context entity is given a standardized label for naming and addressing purposes. The chosen model strongly depends on the system's goals. Ontologies, as defined sets of domain-specific vocabularies, for example, are the most widespread way of abstracting and representing context for smart environment applications, such as EasyMeeting [25]. Sophisticated ontologies are powerful in terms of modeling the relation between contexts and allow rule-based context reasoning. However, there is a trade-off between this functionality and their large overhead.

In this work, a model is needed that allows applications to (i) access current context from sensors, (ii) request predicted context from the responsible component, (iii) influence context via actuators, and (iv) specify their requirements towards their context, without specific knowledge about the individual services in the environment that provide the functionality. In other words, the model has to allow to simply *get* a context (prediction) and *set* a context state, as well as express terms of first-order logic. This is achieved by introducing *context variables* as an abstraction layer for distribution-transparent interaction between applications and their context.

Figure 5.2 illustrates the concept of variable-based context interaction. The idea behind context variables is to extend location models with variables that represent the different types of context that are present at each location, and associate all available services with the respective variables. The concept takes advantage of the location's primary characteristic for context, as well as the typical spatial restriction of a smart environment. That is, applications are typically only dependent on the context at one location at a given point in time. Hence, location can be used as a natural index in order to decrease the complexity in the system. Further, a centralized *context broker* can maintain a service registry that maps individual services to context variables based on their descriptions, and in turn offer the services' capabilities to the applications in a uniform fashion.

Next, the set of queries for communicating with the centralized context broker, i.e., querying context information and instructing context changes, are presented.

5.2.2 Context Queries and Subscriptions

Context queries are one of the key components of context-aware systems. They allow context-aware entities to request the information they need in order to adapt to the environment. In proactive systems, queries are also used to acquire context predictions and instruct context adaptations. Hence, this section will define *context information* as well as *context adaptation queries*, with the former used to obtain information about context variables, while the later group addresses how context variables can be influenced. More precisely, the *context location query* allows an application to acquire the position of an object, while the *context state query* returns information on an object's state, and the *context time query* is used to anticipate when an object will be in a certain state. Finally, the *context adaptation capability query* gives information on which context variable can be influenced, while the *context adaptation instruction query* is used to instruct a context change. In the query descriptions, the term *context configuration* is used to describe a set of context variables and their respective context variable states. As an example, weather conditions are defined by temperature, humidity, precipitation and cloud level. One weather configuration would be 20°C, 65%, slight rain and cloudy. Further, the term *confidence* is used instead of probability, in order to include the quality of measured context, e.g., the reliability of a sensor.

The *context location query* Q^L allows applications to determine at which location a set of context variables are in a specified context configuration during a given time interval, including the location of the requesting application itself.

Definition 5.2.1 (Context Location Query)**. Query** $Q^L = \{V, S, t_1, t_2, P\}, t_1 \leq t_2$ is a 5-tuple, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of context variables, $S = \{s_1, s_2, \ldots, s_n\}$ a set of context variable states, t_1, t_2 denote a timeframe, and $P = \{p_1, p_2, \ldots, p_q\}$ a set of QoS parameters. **Result** $R^L = \{L, \Gamma, \Sigma^2\}$ is a 3-tuple, where $L = \{l_1, l_2, \ldots, l_m\}$ is a set of locations, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ a set of confidences, and $\Sigma^2 = \{\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2\}$ a set of variances.

The *context state query* Q^S aims at the context configuration of context variables with regard to time and location.

Definition 5.2.2 (Context State Query)**. Query** $Q^S = \{V, l, t_1, t_2, P\}, t_1 \leq t_2$ is a 5-tuple, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of context variables, l a location, t_1, t_2 denote a timeframe, and $P = \{p_1, p_2, \ldots, p_q\}$ a set of QoS parameters. **Result** $R^S = \{S, \Gamma, \Sigma^2\}$ is a 3-tuple, where $S = \{s_{1,1}, \ldots, s_{1,n}, \ldots, s_{m,n}\}$ is a $m \times n$ matrix of context variable states, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ a set of confidences, and $\Sigma^2 = \{\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2\}$ a set of variances.

The *context time query* Q^T provides context-aware applications with a prediction, when a specified context configuration will occur at a given location.

Definition 5.2.3 (Context Time Query)**. Query** $Q^T = \{V, S, l, P\}$ is a 4-tuple, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of context variables, $S = \{s_1, s_2, \ldots, s_n\}$ a set of context variable states, l a location, and $P = \{p_1, p_2, \ldots, p_q\}$ a set of QoS parameters. **Result** $R^T = \{T, \Gamma, \Sigma^2\}$ is a 3-tuple, where $T = \{t_{1,1}, \ldots, t_{m,1}, t_{m,2}\}$, $t_{i,1} \leq t_{i,2} \, \forall \, i$, $1 \leq i \leq m$ is a $2 \times m$ matrix of timeframes, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ a set of confidences, and $\Sigma^2 = \{\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2\}$ a set of variances.

The *context adaptation capability query* Q^{AC} provides information about the possibility to influence context variables. The result may vary between simple Boolean values, i.e., true and false, and more complex specifications, such as gradually, infinitely variable or approximately.

Definition 5.2.4 (Context Adaptation Capability Query)**. Query** $Q^{AC} = \{V, l, t_1, t_2, P\}, t_1 \leq t_2$ is a 5-tuple, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of context variables, l a location, t_1, t_2 denote a timeframe, and $P = \{p_1, p_2, \ldots, p_q\}$ a set of QoS parameters. **Result** $R^{AC} = \{A, \Gamma, \Sigma^2\}$ is a 3-tuple, where $A = \{a_{1,1}, \ldots, a_{1,n}, \ldots, a_{m,n}\}$ is a $m \times n$ matrix of adaptation capabilities, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ a set of confidences, and $\Sigma^2 = \{\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2\}$ a set of variances.

The *context adaptation instruction query* Q^{AI}, in contrast to all the others, does not retrieve information or predictions, but initiates a context change. The effect of an issued instruction can be monitored by use of the context state query Q^S. Hence, it does not require a result object.

Definition 5.2.5 (Context Adaptation Instruction Query)**. Query** $Q^{AI} = \{V, S, l\}$ is a 3-tuple, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of context variables, $S = \{s_1, s_2, \ldots, s_n\}$ a set of context variable states, and l a location.

By use of the five queries defined above, an application can request location-, identity-, and time information, query which environment adaptation is possible, and trigger such an adaptation. Hence, it can access predictions on the three primary contexts in order to adapt ahead of time, as well as actuate context.

Rarely are applications interested in a current or predicted context information once at a certain time. Instead, they monitor the information that is relevant to them. To ease the applications' workload, the framework additionally offers to subscribe to context queries. It provides this notification service for context prediction updates, as well as for current context information and context services. Applications – or any other entities in the system – that make use of this service must implement the *subscriber* interface, as specified in Section 5.2.4.

Essentially, subscribers use the same queries as for conventional information requests, but add the desired update period and total lifespan of the subscription. Additionally, entities subscribing to context location and context state predictions can choose between two time-related operation modes. First, the *fixed timestamp mode* provides updates for a prediction that is fixed on a specific time. That is, the time specified in the query remains static in the updates, while the prediction itself and/or its quality may vary. Second, the *leading state mode* offers updates

Method	Query	Parameters						Return	Updates
		v	s	l	t	γ	σ		
`queryLocation`	Q^L	+	+		+			l, γ, σ	
`subscribeLocation`	Q^L	+	+		+			`subID`	l, γ, σ
`queryContext`	Q^S	+		+	+			s, γ, σ	
`subscribeContext`	Q^S	+		+	+			`subID`	s, γ, σ
`queryTime`	Q^T	+	+	+				t, γ, σ	
`subscribeTime`	Q^T	+	+	+				`subID`	t, γ, σ
`queryService`	Q^{AC}	+	+	+	+			CS	
`subscribeService`	Q^{AC}	+	+	+	+			`subID`	CS
`queryAdaptation`	Q^{AI}	+	+	+				`void`	
`reportContext`		+	+	+	+	+	+	`void`	
`cancelSubscription`		`subID`						`void`	

v = context variable, s = context variable state, l = location, t = time,
γ = confidence, σ = variance
CS = Context Service
`subID` = subscription task identification number

Table 5.1: The Context Broker Interface

for predictions in a preceding fashion. That is, the time specified in the query is not fixed but denotes the preceding interval, which in turn stays static, i.e., the time specified in the query is increased every update period by the update period. To illustrate, an example use case for this leading state mode is an application that issues alerts in case it is about to rain in x minutes.

As result to the subscription request, the subscriber receives an identification number for the subscription for future reference. Subsequently, the first update is sent to the subscriber as well, in order to avoid an additional query in case the subscriber needs the information immediately. Finally, updates are only sent in case of changes to preserve bandwidth and resources on the subscriber side.

Next, the interface of the centralized context broker is developed based on the set of context queries.

5.2.3 Context Broker Interface

Table 5.1 presents the interface of the central context broker, including the parameters of the query, the resulting information and, in case of a subscrip-

tion, the information provided by the updates. Results and updates, except for subscription identification numbers, are either embedded into the generic `ContextInformation` object, which consists of one context variable and a dynamic list for each of the other parameter types, or the generic `ContextService` object in case of a service query/subscription. This is done for two reasons. First, the result and update objects are uniform, which allows the use of a two method interface for context subscribers, one method for information updates and one for service updates. Second, and more importantly, the result and update objects may have a different timestamp than the query, either in the case that the context information was retrieved from the database following the *most recent information* strategy, or in the case that the subscription of a context prediction is in the *leading state* mode. In addition to the context queries stated in Section 5.2.2 and the subscription of these queries, the context broker also provides a method to report context, i.e., bring knowledge into the system in a push fashion. A possible scenario for exploiting this functionality are very resource-poor devices, e.g., solar powered sensors, that may not be queried for measurements at will, but rather report the condition of their respective context variable regularly.

Next, the architecture of the context interaction model is described.

5.2.4 Component Architecture

Typically, pervasive systems are designed as *smart environments*, consisting of centralized services on resource-rich machines, statically deployed sensors and actuators, and context-aware applications running on mobile, resource-limited devices. In this work, similar smart environments are assumed, in which context interaction is offered by a centralized brokering entity. However, all context requesting-, providing-, and altering services are considered to be mobile, and they may frequently join and leave the environment (cf. System Model in Chapter 4). Therefore, the context interaction model is built on top of the BASE middleware [11], which is designed for highly dynamic environments featuring a range of resource-limited to resource-rich devices.

Figure 5.3 shows the architecture of the context interaction model. The components are divided into three tiers based on their roles. The *context tier* holds the central context management, as well as sensor- and actuator services, which

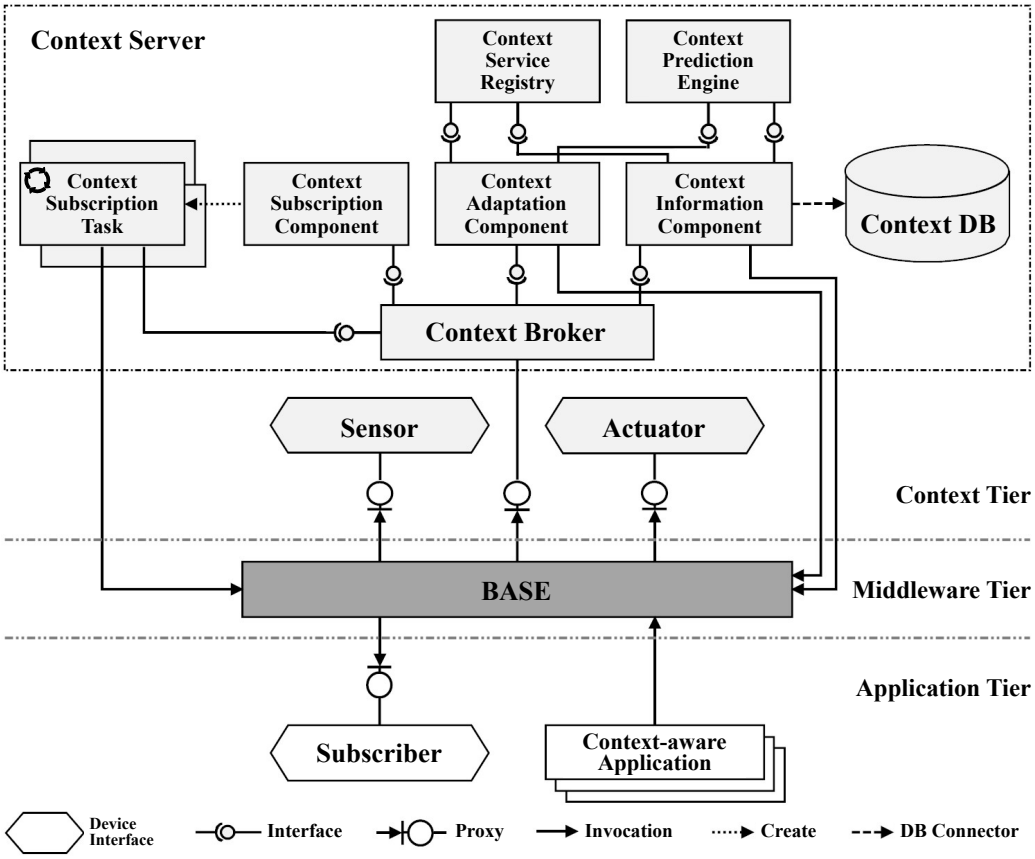


Figure 5.3: Architecture of the Context Interaction Model

are the components directly interacting with the environment's context. The *application tier*, on the other hand, holds the context-aware applications, i.e., the components requesting context interaction. Finally, the *middleware tier*, featuring BASE and the components' respective proxies, handles the environment's devices and the communication between them. In the following, the key components of the context tier are described in more detail.

The *context broker* is the access point of the context management. Context consumers use the context broker proxy to invoke the queries defined in Section 5.2.2, as well as subscribe to these queries. That is, consumers – including context prediction components and the configuration management component presented in Section 5.4 – can request context state-, location-, and time information, as well as context services, either directly or as a customizable subscription. By subscribing to their respective data set, prediction components are notified upon updates and can react to them, e.g., learn and recompute previous predictions.

```
interface Subscriber {
    void update(long taskID, ContextInformation ci);
    void update(long taskID, ContextService cs);
}
```

Figure 5.4: The interface to be implemented by all subscribing entities.

In any case, the context broker distributes incoming queries to the responsible component(s). To do so, the broker dissects the query by examining the specified parameters. During this analysis, the broker also checks for corrupt queries.

Context acquisition and representation is the most basic functionality of a context management in general, and the responsibility of the *context information component*. That is, the component provides access to sensor services as well as predictions on the future availability of sensor services, and administrates the *context database*. The actual procedure of the context information component depends on the time parameter specified in the query it receives. Historic information is directly retrieved from the context database, whereas queries for future information, i.e., context predictions, are forwarded to the *context prediction engine*. For current context, the component triggers – if available – a suitable sensor service. Otherwise, it returns the most recent database entry.

The *context adaptation component* provides information on which context can be adapted at a specified location and time. Hence, it differentiates between queries aimed at current actuation and those requesting future information. For present, i.e., immediate actuation, a simple service look-up via the service registry is sufficient and the component forwards adaptation instruction queries to suitable actuator services. For the availability of future actuator services, the adaptation component requests a location prediction for all suitable services and matches the requested with the predicted location. However, this approach does not consider services entering or leaving the network. Hence, applications should subscribe to this information and take action upon possible updates. In any case, if no suitable service was found, the component returns a *no service available* message to the querying application.

For each subscription, the *context subscription component* creates a *timer task* object as a local representation of the requesting component. The timer task queries the information according to its subscription type and notifies the sub-

```
interface ContextService {
    Location getLocation();
    ContextVariable getContextVariable();
    ContextServiceType getContextServiceType();
}
```

Figure 5.5: The interface that is implemented by all context services.

scriber in case of an update. Figure 5.4 depicts the necessary subscriber interface that the consuming entities need to implement. As described in Section 5.2.2, the subscription type parameter includes update period, total lifespan, and subscription mode. Consumers can either request updates for a static point in time or a dynamic window. The latter mode keeps a static distance between time of subscription and the initially passed time frame. The context adaptation instruction query (Q^{AI}) can not be subscribed, as it represents a single action in the present. Finally, subscriptions can be canceled and context information can be pushed into the system.

All *context services* share a common interface (see Figure 5.5). It provides means to acquire (i) the location of the service, (ii) the context variable associated with it, and (iii) its service type. The broker's context service registry maintains a dynamic directory of all services in the environment by use of that interface. By default, the system features the three service types *sensor*, *actuator*, and *predictor*. However, the list is extensible, as other service types, e.g., *predicting sensors*, are conceivable as well. Chapter 6 discusses the context broker and the other components of the system as it pertains to their implementation details.

The next section presents the application configuration model that allows applications to specify their requirements towards their context – in order for the system to (pre)calculate the adaptation alternatives for the applications based on these requirements – as well as introduces a duration-dependent cost-utility-model as a metric for the quality of an adaptation decision in proactive systems.

5.3 Application Configuration Model

With generic automatic and application-controlled automatic adaptation, an external entity accessible through the underlying system determines the adapta-

tion decision for the application. For this, the deciding entity needs a specification of the application's requirements towards its context. Further, the application should be able to provide additional information on the value of the fulfillment of a requirement, e.g., in the form of user preferences or utility functions, in case there are multiple adaptation options to choose from. This section presents the application configuration model. First, the application requirements definition allows to model the application's context dependencies. The framework later uses these requirements specifications to determine all possible adaptation alternatives of the respective applications. Second, the time-dependent utility and cost model introduces a metric for rating application configurations with regard to the duration the configuration is instantiated. With such a metric, the framework can further optimize adaptation decisions based on adaptation strategies, such as minimal energy consumption, minimal network link loss, or maximal user experience.

5.3.1 Application Requirements

The application requirements model is based on the concept of context variables – as introduced in the previous section – and consists of two steps. The requirements' specification, in turn, is designed to be used as input of the configuration algorithms.

First, the contexts that are relevant for the application are identified, e.g., `VISUAL_OUTPUT` and `LIGHTING`. Second, for each of these variables, those states that are functional for the application are specified. These states may either be defined by symbolic labels, numeric categories, or continuous scales. For the `VISUAL_OUTPUT`, for example, this may be the display size in inches, as well as the resolution either measured in pixels or by labels such as `Full-HD`. For the `LIGHTING`, a certain range on the lumen scale makes sense. Each pair of context variable and its set of valid states is referred to as an *atomic requirement*. In order to check the satisfiability of the set of atomic requirements, they are expressed in boolean logic:

$$\texttt{LIGHTING} \wedge \texttt{VISUAL\_OUTPUT} \tag{5.3.1.1}$$

The conjunction of all atomic requirements of an application describes a *functional configuration* of that application.

In some cases context variables are interchangeable, i.e., an application may either require context `A` or context `B`. This is especially the case if the context variables are related to each other. A navigation application, for example, may use a visual and an audio output, or one or the other. The relation here is that both are output devices. In boolean logic, interchangeability is expressed by disjunction:

$$\texttt{LIGHTING} \wedge (\texttt{VISUAL\_OUTPUT} \vee \texttt{AUDIO\_OUTPUT}) \tag{5.3.1.2}$$

Including these disjunctions, requirements can specify flexible configurations, including using different adaptation techniques for satisfying the same atomic requirement. In the navigation application example, a switch from a visual to an audio output may be both a compositional as well as a behavioral adaptation. The display embedded in the dashboard is needed by a different application, so the navigation application on the user's phone unbinds the display and unmutes the phone's speakers. Flexible configurations can result in multiple adaptation options. However, conditional requirements do not have to be specifically modeled here. Instead, they are implemented through the respective ratings of the possible configurations and the subsequent decision on the best-fit alternative.

A configuration is only functional if all atomic requirements are satisfied. This is best determined by checking the satisfiability of requirements in conjunction. Thus, the set of functional configurations are expressed in *disjunctive normal form* (DNF) – i.e., as the disjunction of the conjunction of the atomic requirements:

$$(\texttt{VISUAL\_OUTPUT} \wedge \texttt{LIGHTING}) \vee (\texttt{AUDIO\_OUTPUT} \wedge \texttt{LIGHTING}) \tag{5.3.1.3}$$

Figure 5.6 depicts the model of the functional configuration of an application. Each application has at least one term made up of at least one atomic requirement, where the atomic requirements are connected through conjunction that form a term, and the terms are connected through disjunction. The atomic requirements are either numeric or symbolic, and are linked via their type to the respective context variable. The requirements' attributes store the current state of the context, as well as a set of states that the application can also use. Overall, the set of requirements form a boolean expression in DNF. As shown in [50], any logic term can be expressed in DNF. Therefore, the requirements model is sufficient in terms of expressiveness.

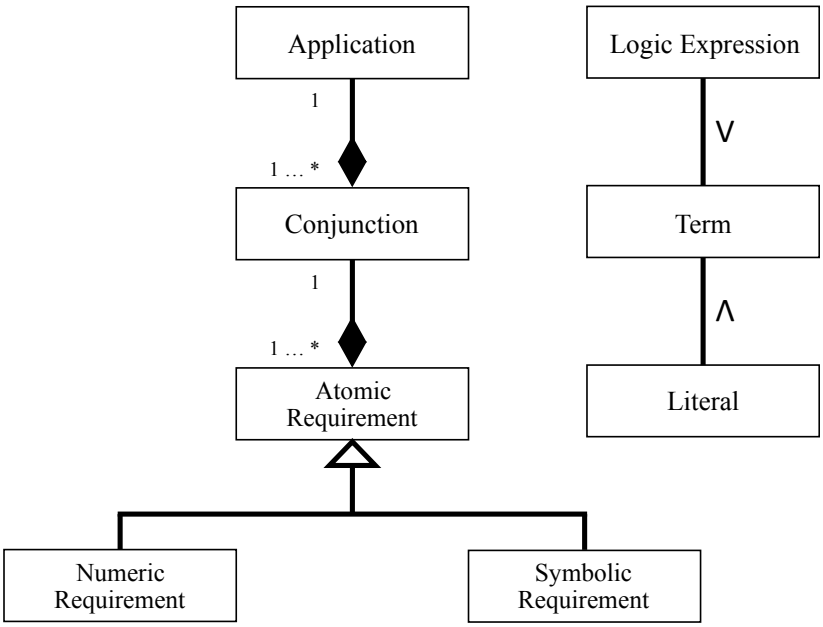


Figure 5.6: Requirements Modeling

The framework's configuration algorithms later on evaluate the boolean expressions in search of possible adaptation alternatives. Hence, the decision how to model the set of requirements has an effect on the algorithms' runtime. Using DNF, the entire set of terms – i.e., the set of functional configurations – can be evaluated in $O(n)$, where n is the number of disjunctions in the set of terms. Equation 5.3.1.3, for example, is evaluated in two steps. First, the algorithm tries to satisfy (`VISUAL_OUTPUT` $\wedge$ `LIGHTING`), and in the second step (`AUDIO_OUTPUT` $\wedge$ `LIGHTING`). The algorithm is discussed in detail in Section 5.4.2.

The next section models the utility and costs of an application configuration with regard to the duration the configuration is expected to be instantiated.

5.3.2 Utility and Cost over Time

The concept of proactive adaptation includes optimizing adaptation decisions based on the expected future. In other words, the goal is not only to pre-compute all possible adaptations, but additionally to determine the adaptation that maximizes the ratio between the utility of the configuration and the costs of switching to and maintaining it, i.e., fixed and running costs. This ratio depends on the duration of the configuration's instantiation. For one, the utility of a configuration

– a numerical value expressing the benefit for the user – may change over time. That is, even though services deliver a consistent quality, they can have different effects on human variables, such as attention span or motivation. However, the duration mainly has implications on the costs part of the ratio. Especially in the context of pervasive computing, where most devices are resource-restricted. That is, a configuration with high fixed costs and low running costs will at some point become more beneficial than one with low fixed costs but high running costs. Taking all this into account, modeling a utility-cost ratio for proactive adaptation is a new challenge.

5.3.2.1 Utility and Preferences

Different users have different work habits as well as preferences, and, therefore, a different perception of a system's utility. They need to be taken into account when comparing an application's configurations. This section presents the modeling of utility functions and user preferences for proactive adaptation. These utility functions and preferences may, for example, be specified by the developer and customized by the user. During runtime, they may also be evaluated and modified, for example using reinforcement learning techniques. However, support for specifying and maintaining utility functions and user preferences is not in the scope of this thesis. Work on how to determine the preferences of a user via graphical user interfaces can, for example, be found in [113].

As described in Section 5.3.1, a configuration consists of a conjunction of atomic requirements towards the state of a context variable, where the state of the context variable and, therefore, the atomic requirement can either be symbolic or numerical, i.e., discrete or continuous, respectively. Hence, the model describing the utility of a configuration based on the preferences of a user must also support both types. First, let $X^{dis} = \{x_1^{dis}, x_2^{dis}, ..., x_n^{dis}\}$ be the set of symbolic context variables and $X^{con} = \{x_1^{con}, x_2^{con}, ..., x_m^{con}\}$ be the set of numeric context variables. Then, D_i^{dis} is the domain of the discrete context variable x_i^{dis} with $D_i^{dis} = \{d_{i,1}^{dis}, d_{i,2}^{dis}, ..., d_{i,k}^{dis}\}$, and D_j^{con} is the domain of the continuous context variable x_j^{con} with $D_j^{con} = [d_{j,min}^{con}, d_{j,max}^{con}]$. With this, it is possible to define the respective discrete or continuous utility functions that return a utility from $[0, 100]$ for any $d_i^{dis} \in D_i^{dis}$ or $d_j^{con} \in D_j^{con}$, respectively. To illustrate, assume there is

the discrete context variable x_{res}^{dis} describing the resolution of a projector and the continuous context variable x_{vol}^{con} describing the volume of an audio output. Then, the utility function for the discrete variable x_{res}^{dis} has the form

$$u_{res}^{dis}(d_{res}^{dis}) = \begin{cases} 10 & \text{if } d_{res}^{dis} = \text{'320p'} \\ 20 & \text{if } d_{res}^{dis} = \text{'480p'} \\ 50 & \text{if } d_{res}^{dis} = \text{'720p'} \\ 100 & \text{if } d_{res}^{dis} = \text{'1080p'} \end{cases} \tag{5.3.2.1}$$

whereas the utility function for the continuous variable x_{vol}^{con} – in this instance using the Gaussian distribution to describe the utility of a volume output – has the form

$$u_{vol}^{con}(d_{vol}^{con}) = (\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(d_{vol}^{con}-\mu)^2}{2\sigma^2}}) * 100,\ \mu \in D_{vol}^{con}. \tag{5.3.2.2}$$

With this and irrespective of the type of variable, let $U(X,t)$ be the set of utility functions for variables X and duration t, with

$$U(X,t) = \{u_{x_1}(d_{1,1},t), ..., u_{x_1}(d_{1,k},t), ..., u_{x_n}(d_{n,l},t)\} \tag{5.3.2.3}$$

such that $u_{x_i}(d_{i,j},t)$ specifies the utility of variable x_i instantiated with state $d_{i,j}$ for a duration of t. Figure 5.7 illustrates such duration-dependent utility functions.

The utility functions determine the utility of a single context variable, respectively the context variables' states. However, users have different perceptions of which aspects of an application are important. In the movie theater, for example, one user might prefer the best acoustic, whereas another user would rather have the most comfortable view. Hence, simply accumulating the single utilities in order to determine the configuration's utility is not a valid approach. Instead, it should be able to attach weights that represent the importance of each aspect. These weights may also change with the duration of the context. Similar to the utility, the equation

$$W(X,t) = \{w_{x_1}(t), w_{x_2}(t), ..., w_{x_n}(t)\} \tag{5.3.2.4}$$

is defined as the set of weight functions, such that $w_{x_l}(t)$ specifies the weight that should be assigned to the utility of variable x_l depending on the duration t of

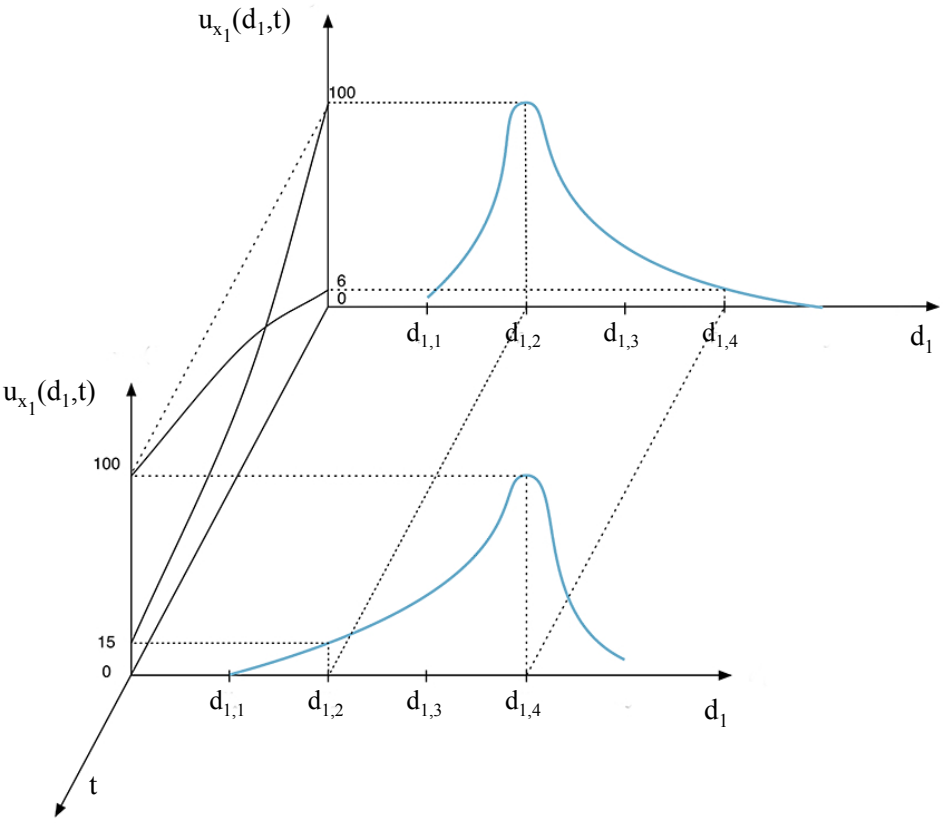


Figure 5.7: Example of Duration-dependent Utility Functions.

the context. The weights are relative. That is, they can be any natural number, they are then divided by the sum of all weights in order to reach an accumulated weight of 1, i.e., $\sum_{l=1}^{n} w_{x_l}(t) = 1$.

Subsequently, the utility of a configuration is the weighted sum of the single utilities. Let $Y = (X, D)$ be a configuration with a set of variables $X = \{x_1, x_2, ..., x_n\}$ instantiated with the corresponding states in domain $D = \{d_1, d_2, ..., d_n\}$. Then, the utility of configuration Y for duration t is

$$u(Y,t) = \sum_{i=1}^{n} (w_{x_i}(t) * u_{x_i}(d_i,t)). \tag{5.3.2.5}$$

Together, the set of utility and weight functions then form the preferences of a user. Accordingly, $P(X,t)$ specifies a preferences description, with

$$P(X,t) = \{U(X,t), W(X,t)\}. \tag{5.3.2.6}$$

5.3.2.2 Costs

As mentioned above, the costs of an adaptation are a very important aspect when dealing with resource-restricted devices. These resources, such as CPU, memory, bandwidth, and battery, are the predominant cost factors. They are related to the level of service the application offers its user. People commonly use a more energy-efficient setting of their laptops in mobile scenarios, or switch off mobile network connectivity in order to save their phones' battery. This work assumes that the applications have knowledge about their different resource profiles and the profiles of the services they use, as well as which level of service, i.e., utility, they provide with them.

However, there are other cost factors in pervasive environments besides resources. [14] identifies the time it takes to adapt as costs. They depend on the complexity of the adaptation. For example, a behavioral adaptation, e.g., switching to silent mode, is less complex than a compositional one, e.g., switching an I/O device. Hence, the time needed for an adaptation depends on the previous configuration instantiation of an application. Typically, the similar the previous configuration is, the less complex the adaptation. Additionally, [3] identifies the distraction of the user due to an adaptation as costs. The user is not only forced to pause during, but also needs effort on his/her part to cope with the adaptation, e.g., reorientation. However, this cost factor varies from user to user and can not be measured easily.

In order to compare the costs of various adaptation alternatives over the duration of a context, it is necessary to be able to specify the fixed costs for switching to, as well as the running costs of maintaining the new configuration. In case new services become available, they can then be compared to the current configuration by considering the current configuration to be an adaptation with zero

fixed costs. Equation

$$c(Y_{new}, Y_{old}, t) = c_{fix}(Y_{new}, Y_{old}) + c_{run}(Y_{new}, t) \tag{5.3.2.7}$$

defines the cost function, with $c_{fix}(Y_{new}, Y_{old})$ as the fixed transition costs from the current to the new configuration, and $c_{run}(Y_{new}, t)$ as the running costs of the new configuration depending on the duration t of the new context. The fixed transition costs $c_{fix}(Y_{new}, Y_{old})$ depend on the resources spent, the time needed for the adaptation, and the distraction of the user. Each of these factors are difficult to estimate and, thus, should best be learned by the specific system. The running costs $c_{run}(Y_{new}, t)$, on the other hand, are easier to handle. They consist only of the resources the application consumes in the respective configuration. As described above, each application is able to provide their resource profile as well as that of the services they use. However, the different resources are measured using different units, e.g., *clocks* for CPU and *bytes* for memory. In order to determine one accumulated running costs value, it is necessary to normalize them to a value in $[0, 100]$. That is, assuming there are three alternative configurations with running CPU costs $c_{CPU_1} = 5560$ *clocks*, $c_{CPU_2} = 6280$ *clocks*, and $c_{CPU_3} = 1200$ *clocks*. Then it is possible to determine $c_{CPU_{max}} = 6280$ *clocks*, and normalize all costs with $c'_{CPU_i} = c_{CPU_i} * \frac{100}{c_{CPU_{max}}}$. The normalized costs are then $c'_{CPU_1} = 88.54$, $c'_{CPU_2} = 100$, and $c'_{CPU_3} = 19.10$. After doing the same with other resources, it is possible to accumulate all running costs to one value using

$$c_{run}(Y_{new}, t) = \sum_{i=1}^{n} c'_{y_i} * t. \tag{5.3.2.8}$$

In order to adjust the function to the devices characteristics, the normalized costs of the different resources can additionally be weighted, e.g., using the amount of the total or currently available resources of the device.

The next section combines the utility and cost functions to cost-utility ratios.

5.3.3 Duration-dependent Cost-Utility-Ratios

After the configuration algorithms have determined all adaptation alternatives for an application, i.e., its set of functional configurations that are viable in the

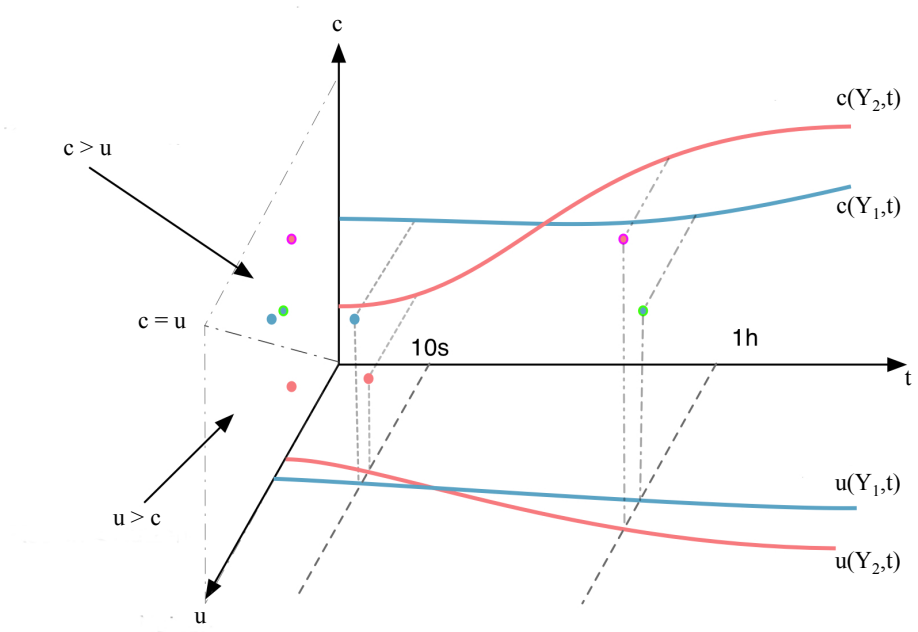


Figure 5.8: Example of Duration-dependent Cost-Utility-Ratios.

predicted context, it is necessary to decide on which configuration to instantiate. The previous sections presented an approach to modeling utility and costs of a configuration considering the duration the configuration will be active. This section shows how to apply the cost and utility functions to a set of alternative configurations.

Figure 5.8 illustrates such a rating for the two alternative configurations Y_1 and Y_2, with the costs on the y-axis, the utility on the z-axis, and the duration of the context on the x-axis. In this example, alternative Y_1 has the higher fixed costs, but Y_2 becomes more expensive after a certain point due to its higher running costs. The same is true for the utilities of Y_1 and Y_2. Hence, the factor determining which alternative has the higher or lower costs/utility depends on the expected duration. With the cost and utility functions, it is possible to determine a cost-utility-ratio for each configuration and any expected duration(s), indicated in the figure using colored dots. To further illustrate this ratio, the dots are projected to the area between the y and the z-axis. Splitting the area into two triangles with $c = u$ distinguishes between cost-utility-ratios ($\frac{c}{u}$) of 1, > 1, and < 1. Naturally, it is preferable to minimize this ratio, getting the most utility per

cost. However, there exist scenarios in which the user might, for example, define a lower boundary for the application's utility and accept a higher ratio in order to exceed that boundary. As shown in Section 5.4.3, the rated configurations are therefore forwarded to a decision component that applies certain user-specific adaptation strategies.

As both utility and costs are modeled as functions, rating a single context variable assignment happens in constant time $O(1)$. A configuration Y is a set of context variables instantiated with a state from its domain. Hence, rating a configuration is linear in the number of its context variables, i.e., $O(|Y|)$. In order to make a decision, it is necessary to rate all configuration alternatives. Let $Z = (Y_1, Y_2, ..., Y_p)$ be the set of alternative configurations and $|Y|_{max}$ be the maximum size of any $Y_q \in Z$. Then, the complexity of rating all alternative configurations is $O(|Z||Y|_{max})$.

The next section presents the configuration algorithms that find all possible adaptation alternatives of an application given a certain context. Afterwards, Section 5.5 shows how to coordinate the adaptation of multiple applications in a shared environment, before Section 5.6 presents the adaptation control loop.

5.4 Comparable Adaptation Alternatives

This section presents the configuration algorithms that search for all possible adaptation alternatives of an application based on the current or predicted context in the environment. That is, for any given context – i.e., the physical conditions and available services at a location – the algorithms search for all valid configurations. For this, the problem of finding adaptation alternatives is formulated as a constraint satisfaction problem, and subsequently solved using a backtracking-based algorithm, an ordering heuristic, as well as a context service index structure. Finally, the section gives an architectural overview of all components involved in searching for and rating adaptation alternatives.

5.4.1 Application Configuration as Constraint Satisfaction

The validity of an application configuration is determined by the application's requirements and the environment's adaptation capabilities. Both set constraints

towards the search space. First, a configuration must satisfy all application requirements. Second, the environment must provide the necessary resources and adaptation capabilities to satisfy the configuration. Hence, the problem of finding a valid application configuration is a constraint satisfaction problem.

Based on [94], a *constraint satisfaction problem* (CSP) is a triple (V, D, C), where $V = \{V_1, \ldots, V_n\}$ is a finite set of variables and $D = \{D(V_1), \ldots, D(V_n)\}$ is a set of finite domains, such that $D(V_i)$ is the finite set of potential values for V_i. Furthermore, $C = \{C_1, \ldots, C_x\}$ is a finite set of constraints, where each C_y is a pair (t_y, R_y) with $t_y = (v_{y_1}, \ldots, v_{y_z})$ being a z-tuple of variables and R_y being a z-ary relation over D. A solution of an instance of a CSP is a function $f : V \rightarrow D$ such that $\forall (t_y, R_y) : f(t_y) = \{f(v_{y_1}), \ldots, f(v_{y_z})\} \in R_y$.

The adaptation capabilities of the environment can differ at each location. For example, a meeting room usually has a projector, whereas a typical office does not. However, these capabilities constitute the variables and respective domains of the CSP. Hence, the CSP has to be constructed and solved for each location individually. With this in mind and the previous definition, the problem of finding a valid application configuration for a single location can be modeled as the following CSP:

Let L be the finite set of locations $L = \{L_1, \ldots, L_p\}$ in the environment. Then, let V^{L_q} be the finite set of context variables $V^{L_q} = \{V_1^{L_q}, \ldots, V_n^{L_q}\}$ at location L_q, and let D^{L_q} be the set of finite domains $D^{L_q} = \{D^{L_q}(V_1^{L_q}), \ldots, D^{L_q}(V_n^{L_q})\}$, where $D^{L_q}(V_i^{L_q}) = \{S_{i_1}^{L_q}, \ldots, S_{i_m}^{L_q}\}$ is the finite domain of $V_i^{L_q}$, namely the finite set of possible context variable states ($S_{i_j}^{L_q}$) for $V_i^{L_q}$. Further, $C = \{C_1, \ldots, C_x\}$ is a finite set of constraints, namely the set of terms in the application's requirements, where each C_y is a pair (t_y, R_y) with $t_y = (v_{y_1}, \ldots, v_{y_z})$ being a z-tuple of variables – i.e., the context variables in the respective term – and R_y being a z-ary relation over D^{L_q} – i.e., a specific instantiation of the context variables. Finally, a solution to the problem of finding a valid application configuration is an instantiation of each context variable in the application's requirements' term with a context variable state that is both permitted by the constraints, and possible for the environment to provide, i.e., in the set of domains of the location.

The next section presents the adaptation alternative search algorithm that finds all possible adaptation alternatives for every location in the environment.

5.4.2 Adaptation Alternative Search Algorithm

There are various approaches to solving CSPs, e.g., the set of hybrid *backtracking*-based algorithms by Prosser [86] or variations of local search algorithms [94]. The choice, which approach to use, depends on the specific characteristics of the problem, as well as the desired result. For example, Prosser's algorithms aim at finding a solution as fast as possible, whereas local search algorithms are used for finding a near optimal solution without traversing the entire search space (cf., for example, *hill climbing* or *simulated annealing*). The goal in this work is to select the best possible adaptation option ahead of time, coordinated for all applications in the shared environment, using the utility and cost metrics. As a result, the entire set of possible adaptations for any given situation should be readily available, in order to (i) keep the adaptation delay minimal, as well as (ii) provide all adaptation options of every application to the adaptation coordination algorithms (see Section 5.5). Otherwise, possible adaptation plans could be forestalled. Hence, it is necessary to find all possible solutions, instead of terminating after a solution, or a subset of solutions was found. Moreover, as all solutions are to be found, the individual ratings of the solutions are irrelevant during the search – i.e., solution optimization techniques do not apply – and, therefore, can be calculated afterwards.

Next, a backtracking-based adaptation alternative search algorithm is presented, which conducts an exhaustive search, i.e., finds all possible solutions to the problem. Afterwards, an *ordering heuristic* is introduced, which optimizes the search sequence during the search process by calculating a *complexity index* for each context type, and the *context service index structure* is described, which is used to minimize the number of context service queries posed to the context management component during the search.

5.4.2.1 Exhaustive Search

Due to the individual composition of the context at each location, it is necessary to solve one CSP per location, and gradually obtain all solutions from the environment. That is, the search is conducted in p iterations, where p is the number of locations in the environment (cf. the definition of the search problem

in Section 5.4.1). At the start of each iteration, the algorithm replicates the context situation of the location in question, by initializing the set of variables at that location and their respective domains, and solves the CSP. The set of constraints, however, is constant during the whole search process, as the application's requirements are location-independent.

The search space of the CSP of a location can be thought of as a tree. Each level of this tree structure describes a context variable at the given location, and the nodes on that level represent the variable's domain, i.e., the states that the context can become. Then, a possible combination of all context instantiations is a path from the root node to a leaf of the tree. However, not all context variables in the tree are relevant for the application. That is, if the variable in question is not contained in the application's requirements, its instantiation has no influence on the validity of the combination. To avoid unnecessary steps, those irrelevant context variables can be bypassed by removing them from the set of variables V^{L_q} of the CSP prior to the search iteration regarding location L_q. Moreover, not every location in the environment may have the necessary context services in order to satisfy the application's requirements. For example, if the application needs a visual output, such as a projector or a public display, and the location in question can not provide such a component, it is automatically eliminated from the solution space. Hence, such locations can also be bypassed. In any case, if a given combination is valid, i.e., it satisfies all the constraints posed by the application's requirements, it is called *configuration* or *solution*.

The exhaustive search is conducted by a *backtracking*-based algorithm, more specifically a *depth-first search*-based (DFS) algorithm, as shown in Algorithm 1. The algorithm builds all possible combinations of acceptable context states that are feasible for the context services at the location in question. That is, for each location and requirement term, the algorithm conducts a depth-first search on the respective domains of the local context variables. As discussed above, variables that are irrelevant for the application can be bypassed, and locations that do not offer adaptation capabilities for all variables in the requirement term can not provide solutions. Hence, only the domains of those variables are used, which are also contained in the term (cf. Line 8), and the search skips the current location and term pairing, in case the resulting number of domains is less than the number of variables in the term (cf. Lines 10-12). The algorithm starts at the root of

Algorithm 1 Exhaustive DFS-based Configuration Search

```
 1: procedure E-DFS(appID)
 2:     solutions ← null
 3:     locations ← getLocationsInEnvironment()
 4:     appReqs ← getApplicationRequirements(appID)
 5:     for all location ∈ locations do
 6:         for all term ∈ appReqs do
 7:             status ← "unknown"
 8:             domains ← getDomainsInTerm(location, term)
 9:             n ← getNumberOfLiterals(term)
10:             if domains.size < n then
11:                 status ← "impossible"
12:             end if
13:             consistent ← true
14:             i ← 1
15:             while status = "unknown" do
16:                 if consistent then
17:                     i ← dfs-label(i, consistent)
18:                 else
19:                     i ← dfs-unlabel(i, consistent)
20:                 end if
21:                 if i > n then
22:                     i ← i - 1
23:                     solutions.add(getHeadsOfCurrentDomains())
24:                     currentDomains[i].removeHead()
25:                     consistent ← currentDomains[i] ≠ null
26:                 else if i = 0 then
27:                     status ← "known"
28:                 end if
29:             end while
30:         end for
31:     end for
32:     return solutions
33: end procedure
```

the tree and proceeds towards its leaves by *labeling* its nodes (see Algorithm 2). For this, there are the two sets named *domain* and *current domain*. The domain is the entire set of possible values per context variable and remains constant, whereas the current domain is a subset that the algorithm operates on. In case a value in the current domain of a variable is acceptable for the application, i.e., the value is contained in the requirement term, the algorithm has found a new *partial solution*, and continues with the next variable. Otherwise, the algorithm backtracks by *unlabeling* the current node, i.e., it restores the current domain of that variable and continues the search at the preceding level (see Algorithm 3). During the search, the algorithm stores all valid configurations (cf. Algorithm 1,

Algorithm 2 Exhaustive DFS Label

```
1: procedure E-DFS-LABEL(i, consistent)
2:     consistent ← false
3:     while currentDomains[i] ≠ ∅ ∧ ¬consistent do
4:         consistent ← inTerm(currentDomains[i].head)
5:         if not consistent then
6:             currentDomains[i].removeHead()
7:         end if
8:     end while
9:     if consistent then
10:        return i+1
11:    else
12:        return i
13:    end if
14: end procedure
```

Algorithm 3 Exhaustive DFS Unlabel

```
1: procedure E-DFS-UNLABEL(i, consistent)
2:     h ← i-1
3:     currentDomains[i] ← domains[i]
4:     currentDomains[h].removeHead()
5:     consistent ← currentDomains[h] ≠ null
6:     return h
7: end procedure
```

Lines 21-25). As soon as the algorithm returns to the root of the tree, it has explored the entire solution space, and continues with the next term and/or location, until it terminates.

5.4.2.2 Ordering Heuristic

It is possible that specific requirements of applications can only be satisfied in a few locations in the environment. If a single requirement literal of a requirement term can not be satisfied at a location, it is unnecessary to continue the configuration search for that term and location. Hence, such requirements should be checked first, in order to recognize that there are no possible solutions at the location in question, as early as possible. The *ordering heuristic* uses a *most constraint variable heuristic*-based approach (cf. [94]) to optimize the search sequence for the application's requirements. That is, the literals in a requirement term are ordered such that the literals that can most likely not be satisfied by the current location's context resources, are checked first. As a result, the validation process for the current location terminates faster. The earlier a non-fitting loca-

tion is excluded from the search space, the lower is the arising overhead through pursuing non valid locations.

The ordering heuristic needs a metric in order to compare the requirements with each other, and determine in which order they should be processed. Therefore, a so-called *complexity index* is calculated for each context variable in the application's requirements, representing their respective potential to diminish the search space. That is, the complexity index estimates the success rate of the consistency check by comparing the respective sizes of the sets of context states that are acceptable for the application, and those providable by the environment, assuming a uniform distribution of the likelihood of each context state being both desirable and feasible. For comparability, the index has to be bound to a certain range. Assuming that the context states in the applications' requirements are a subset of the possible states in the environment, the quotient of the number of acceptable and possible states is a float number in the interval (0,1], with a low complexity index suggesting a strong potential for reducing the search space. Even though this assumption typically holds true, it must not always be the case. However, as the heuristic tries to eliminate search space by checking the most restricted variables first, i.e., those with a low complexity index, the index can simply be forced into the (0,1] interval, without losing its effect.

When calculating the complexity index, the heuristic has to distinguish between symbolic and numeric requirements. The complexity index k^{sym} of a symbolic context variable V_i^{sym} is the quotient of $a_{V_i^{sym}}^{sym}$ and $u_{V_i^{sym}}^{sym}$, where $a_{V_i^{sym}}^{sym}$ is the number of states of V_i^{sym} that are acceptable for the application, i.e., the states specified in the requirements and constituted in the constraints, and $u_{V_i^{sym}}^{sym}$ is the number of states in the universe of V_i^{sym}, i.e., the size of the union of states for variable V_i^{sym} that are possible in the environment.

$$k_{V_i^{sym}}^{sym} = min(\frac{a_{V_i^{sym}}^{sym}}{u_{V_i^{sym}}^{sym}}, 1) = min(\frac{|v_{ij}^{sym} \in \bigcup_{w=1}^{x} R_w|}{|\bigcup_{q=1}^{p} D^{L_q}(V_i^{sym})|}, 1) \tag{5.4.2.1}$$

In case of a numeric requirement, the required and the possible states of a context variable V_i^{num} are defined by value ranges, instead of sets of discrete values. The relation between two such value ranges is defined by their respective sizes or diameters ($\varnothing$), i.e., the absolute differences between the respective end-

points. Hence, the complexity index k^{num} of a numeric context variable V_i^{num} is calculated by dividing $a_{V_i^{num}}^{num}$ by $u_{V_i^{num}}^{num}$, where $a_{V_i^{num}}^{num}$ and $u_{V_i^{num}}^{num}$ are the absolute values of the distances in the value ranges of the required and the possible values for the numeric variable V_i^{num}, respectively.

$$\begin{aligned} k_{V_i^{num}}^{num} &= min(\frac{a_{V_i^{num}}^{num}}{u_{V_i^{num}}^{num}}, 1) = min(\frac{\varnothing\{v_{i_j}^{num} \in \bigcup_{w=1}^{x} R_w\}}{\varnothing \bigcup_{q=1}^{p} D^{L_q}(V_i^{num})}, 1) \\ &= min(\frac{|max(\{v_{i_j}^{num} \in \bigcup_{w=1}^{x} R_w\}) - min(\{v_{i_j}^{num} \in \bigcup_{w=1}^{x} R_w\})|}{|max(\bigcup_{q=1}^{p} D^{L_q}(V_i^{num})) - min(\bigcup_{q=1}^{p} D^{L_q}(V_i^{num}))|}, 1) \end{aligned} \tag{5.4.2.2}$$

Finally, in order to use the ordering heuristic during the search, Algorithm 1 additionally gets Line 4a `orderByComplexityASC(appRegs)`, which orders the set of requirement literals in each of the requirement terms by their respective complexity index in ascending order. The order, in which the application terms are iterated through, remains irrelevant, as they are independent from each other and must all be accounted for, in order to not forestall any possible solutions during the adaptation coordination phase.

5.4.2.3 Context Service Index Structure

The search algorithm uses a *context service index structure* to minimize the communication overhead during the search process. The index structure can be thought of as a basic context service registry. For each location, it stores all necessary information regarding the environment's context services that is relevant for the search process. This includes information about the location's context service types, the sensor/actuator identification information for measuring and performing environment changes, as well as the so-called *actuator capabilities* that define the states or value ranges, respectively, that the actuators provide. With this information, the algorithm is able to calculate all solutions without any additional data, instead of constantly having to query the context management component for the needed information.

However, using such an index structure requires a constant maintenance effort. In order to assess whether the communication overhead to maintenance trade-off

is beneficial, it is necessary to compare the respective data load that must be transferred to support the search algorithm. The individual data load is coherent to the frequency, in which the algorithm searches for adaptation alternatives, and the amount of environment information data that is necessary for one search. That is, a high frequency of searches, e.g., due to a highly dynamic environment, suggests a positive effect of the index structure. Hence, a suitable metric for estimating the dynamism of the system is the number of executed adaptation calculations over time. On the other hand, the smaller the environment is – in terms of the number of locations in it – and the lower the context service density in the environment is, the lower is the amount of environment data needed for one search. In this case, the constant maintenance effort might be disadvantageous. Even though it is possible to reason about the data load, it is necessary to say that both metrics depend entirely on the actual environment.

There are four aspects that affect the data load of an adaptation alternative search:

1. the size of the complete environmental service information $c_{complete}$,
2. the size of a single context service information update c_i,
3. the number of adaptation calculations in a certain period of time a_t, and
4. the number of context service updates in the same period of time u_t.

Using these factors, the data load of the naive algorithm without the index structure (X_{naive}) is described in Equation 5.4.2.3, and the data load of the algorithm with the index structure (X_{index}) in Equation 5.4.2.4.

$$X_{naive} = c_{complete} \cdot a_t \tag{5.4.2.3}$$

The data load for the naive algorithm is the product of the complete context service information $c_{complete}$ and the number of adaptation calculations in a certain period of time a_t.

$$X_{index} = c_i \cdot u_t + c_{complete} \tag{5.4.2.4}$$

For the index structure-based algorithm, the data load depends on the product of the number of updates regarding the context services in the environment per time unit u_t and the size of a single context service update message c_i. As an initial set of data is needed, the size of all context service information $c_{complete}$ is added.

Assuming $c_{complete}$ is N-times the size of c_i, with N being the number of context services in the environment, it is possible to approximate equation of the two data loads if $N \cdot a_t \approx u_t$, as shown in Equation 5.4.2.5. Hence, if the number of context service updates in a certain period of time is lower than N-times the number of adaptation calculations in the same period – a reasonable assumption considering a typical environment will feature dozens of context services – then the communication overhead to maintenance trade-off of the context service index structure is beneficial in favor of the index.

$$\begin{aligned} X_{naive} &= X_{index} \\ c_{complete} \cdot a_t &= c_i \cdot u_t + c_{complete} \\ c_{complete} \cdot (a_t - 1) &= c_i \cdot u_t \\ N \cdot c_i \cdot (a_t - 1) &= c_i \cdot u_t \\ N \cdot (a_t - 1) &= u_t \\ N \cdot a_t &\approx u_t \end{aligned} \tag{5.4.2.5}$$

To summarize, the algorithm using the index structure has an advantage in large environments with a large number of context services, and in case of many adaptation alternative search runs. Not using the index structure is only superior in environments with many context service updates, but very few search runs. The performance of the variations of the algorithm during simulation is presented and discussed in Chapter 7.

5.4.3 Component Architecture

Figure 5.9 shows the architecture of the configuration management component in the context of the overall system architecture. Above the component are the pervasive applications that make use of its capabilities. The applications register at the component with their respective set of requirements and preferences via the component's interface (see Figure 5.10). In turn – disregarding adaptation coordination for now – the applications receive their rated adaptation alternatives from the component, whenever their current or predicted context changes (cf. Section 5.3). Below is the context management, which delivers context sensor

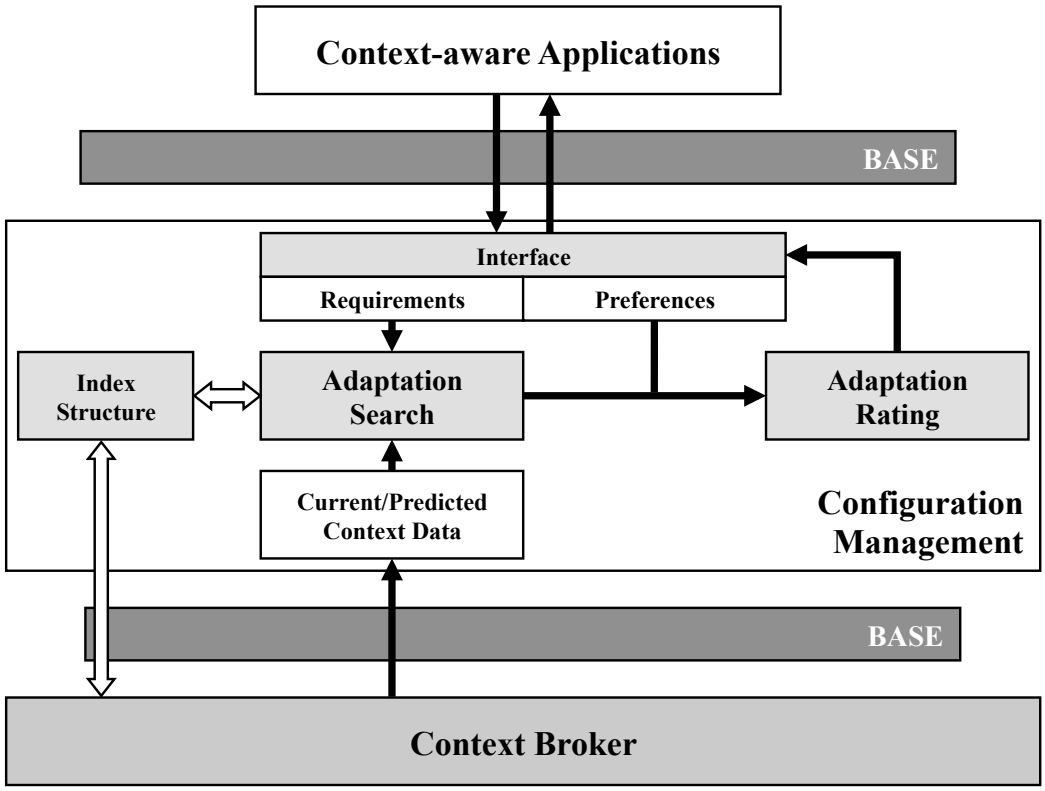


Figure 5.9: Architecture of the Configuration Management Component

and actuator information on demand or by subscription (cf. Section 5.2). Specifically the subscription service of the context management component is used for keeping the index structure up to date. Whenever a context service enters or leaves the network, or changes its location, the configuration management component receives a notification and can update the context service index structure accordingly.

The component's workflow is as follows. It first analyzes the requirements and preferences of the application. Thereupon, the system determines the possible configurations based on the requirements and the (predicted) context information, which is received through the context data component. After that, the resulting configurations are forwarded to the configuration rating component, which also communicates with the context data component in the same manner. Additionally, the configuration rating component receives the analyzed preferences of the context-aware application. Based on these inputs, the configuration rating component determines the best adaptation alternative based on their re-

```
interface ConfigurationManagement {
     long registerApplication(
            ApplicationRequirements appReqs,
            UserPreferences userPrefs,
            ReferenceID refID);
}
```

Figure 5.10: The Interface of the Configuration Management Component

spective utility and costs, as well as the preferences of the user regarding an adaptation strategy. The decision is returned to the context-aware application, which executes the adaptation.

The next section presents the adaptation coordination algorithms that find so-called adaptation plans for multiple applications that share their context, in order to prevent oscillating adaptations and optimize the global utility of the environment. Afterwards, Section 5.6 combines the individual components into the overall adaptation control loop.

5.5 Coordinated Adaptation Plans

Adaptation in multi-user environments requires coordination, as isolated adaptations of single applications that share context often leads to so-called *context interferences*, which force subsequent re-adaptations and, thus, lead to oscillating effects. With proactive adaptation, such interferences would further negate any benefits of pre-calculating adaptation alternatives. In order to avoid such situations, the set of predetermined adaptation alternatives should be coordinated beforehand as well. That is, after each calculation of adaptation alternatives, the system should find all interference-free combinations of adaptation alternatives – referred to as *adaptation plans* – for the location, for which the adaptation alternatives were calculated.

The COMITY framework [63, 64, 65] offers interference detection and resolution for applications in multi-user environments. However, it is not sufficient to run the COMITY framework in parallel for two reasons. First, it is a *reactive* adaptation coordination framework, which means it monitors the active functional configurations of the applications in the environment, and only coordinates the applications in the case an interference actually occurs, instead of

preventing them. Second, COMITY's interference resolution algorithm terminates as soon as an interference-free adaptation plan was found, without regard to the utilities of the individual configurations of the applications. Hence, in order to coordinate the adaptation alternatives that were calculated by the configuration management component ahead of time, while optimizing the global utility in the environment, a new set of tree-based resolution algorithms are developed for this framework, which use COMITY's interference detection to check the validity of the adaptation plans.

This section first gives an overview of the COMITY framework, including a brief discussion of context interferences, the concept of *context contracts*, as well as the interference detection and resolution approach. Afterwards, it presents the set of tree-based resolution algorithms that optimize the global utility of the environment. Finally, the section shows how to derive context contracts from application requirements, in order to be able to utilize the interference detection approach. The integration of adaptation coordination into the configuration management are part of Section 5.6, which presents the overall adaptation control loop.

5.5.1 Adaptation Coordination with COMITY

COMITY is a middleware-based adaptation coordination framework for pervasive environments. The applications in the environment submit their context influences and dependencies to the coordinator – similar to the way they submit their requirements and preferences to the configuration management component in this work – and the coordinator monitors for and resolves interferences in the shared context.

5.5.1.1 Interferences

The parallel execution of pervasive applications poses challenges in multi-platform pervasive systems. The problems arise from the fact that pervasive applications interact with a shared context. Consider the situation in which application App_i has changed the context according to its needs. Afterwards, application App_j is started and discovers that the shared context does not satisfy

its requirements. Consequently, App_j also adapts the context according to its needs. This action changes the basis on which App_i chose its active functional configuration. Since its current configuration is not anymore viable in the changed context, App_i is forced to react, leaving it with two options: It can adapt the context again according to it's own needs or it can adapt itself. The first option may result in a cycle where the two applications take turns adapting the context, which results in an oscillation of the context between two states and with each change one of the applications is not able to operate – comparable to a deadlock. The second option may prove to be suboptimal because another configuration may not satisfy the user's requirements. Moreover, it may be possible that no viable functional configuration can be found at all for App_i.

The problem described above – an application-induced context that in turn forces other application(s) to adapt – is referred to as an *interference*. Interferences occur when applications make configuration decisions, and as a consequence thereof adapt the context according to their needs, without considering that other applications may be executed in parallel.

5.5.1.2 Context Contracts

A *context contract* defines the interaction of an application with its context depending on a functional configuration. That is, a context contract is the specification of all context relations – dependencies and effects – of a functional configuration. For application coordination, each application provides the *active context contract* for its current configuration, and at least one *alternative context contract* for alternative functional configurations. The active context contract is required for interference detection, whereas the list of alternative context contracts is used for interference resolution. Context contracts consist of two mandatory parts, the *interference specification* and the *context influences*.

The first part of a context contract is the *interference specification*. It defines the context states that pose an interference for an application. Figure 5.11 shows an example interference specification of a presentation application. The interference specification models three context states that the application encounters as an interference. The first one is a temperature that is below 19 °C. The second specifies an interference, if a presentation takes place in the environment and the

```
BEGIN
       temperature < 19.0 Celsius
   OR
       activity = video_presentation AND light.intensity != dimmed
   OR
       activity = video_presentation AND audio.volume > 55 decibels
END
```

Figure 5.11: An Example Interference Specification

lights are not dimmed. The third one models an interference when a presentation is held and the audio volume is greater than 45 decibels. As with the application requirements, the interference specifications of an application are defined by the application developer.

The second part of the context contracts are the application's *context influences*. Context influences explicitly specify the effects an application has on the shared context. Figure 5.12 shows an example context influence definition of a video presentation. The context influences state that the application sets the activity of the environment to video presentation, dims the lights, and outputs speech with an intensity of 55 decibels.

Next, COMITY's interference detection and resolution approaches are briefly explained, before the interference resolution algorithm that maximizes the environments utility is presented.

5.5.1.3 Interference Detection and Resolution

In COMITY, the interference detection process is triggered every time there is a change to the set of interference specifications or the context itself. It evaluates all active interference specifications with regard to the current context. In case an interference specification is satisfied, a description of the interference is composed. The description includes the satisfied interference specification, the contributing context and all applications that are involved. Once the description is created, the interference resolution process is triggered.

Interference resolution in COMITY is a two-staged process. First, an *interference resolution plan* is computed, which determines how applications have to adapt in order to resolve a detected interference. If each application fulfills its

```
CI = {activity = video_presentation,
      light.intensity = dimmed,
      audio.type = speech,
      audio.volume = 55 decibels}
```

Figure 5.12: An Example Context Influence Definition

respective assignment, the interference is resolved and an interference-free system state emerges. In order to determine such a plan, the interference resolution component searches for a context contract for each application, such that the detected interference is resolved and no new interferences are created. That is, an assignment needs to be found for each application such that (i) the context influences of the application do not satisfy any then active interference specification – which can change as part of the resolution plan – and (ii) the new interference specification of the application is not satisfied by the then existing context. As with the adaptation alternative search, the problem of finding such a resolution plan is modeled as a CSP. The original COMITY resolution algorithm finds a resolution plan with the minimal number of necessary adaptations. It starts from the last functional set of context contracts, and appends the ones causing the interference at the end. The algorithm then alternates the context contracts of those applications first, which caused the interference. If this first step is not successful, the algorithm uses a *backtracking*-based approach – combined with a pruning technique similar to *backjumping* – in order to find a resolution plan. Once a resolution plan has been obtained, the COMITY framework instructs applications to adapt according to the plan.

Next, the tree-based interference resolution algorithms for proactive adaptation coordination are presented.

5.5.2 Tree-based Interference Resolution Algorithms

A resolution plan with minimal adaptations is desirable, when there is an actual interference in the system. However, this is not the case during pre-coordination of adaptation alternatives, eliminating the starting point of COMITY's resolution algorithm. Further, the approach is not consistent with the concept of proactive adaptation, of which the goal is to predetermine all alternatives in order to be prepared and make the best possible decision. This

section presents a set of tree-based algorithms for finding an interference resolution plan. First, it describes a resolution algorithm that implements the *explicit forward checking with conflict-directed backjumping* strategy [86], which makes use of information gained in the process of finding subsolutions. Afterwards, it presents a modification of that algorithm that does not terminate after finding a solution, but continues the search process to find the optimal solution, while pruning the search space using the *branch and bound* strategy, in order to improve its runtime.

5.5.2.1 Interference Resolution as Constraint Satisfaction

As defined in [63], the problem of computing an interference resolution plan can be modeled as a constraint satisfaction problem as follows:

Let V be the set of applications $App = \{App_1, \ldots, App_n\}$ which are active in the environment, and let D be the set of finite domains $CC(App) = \{CC(App_1), \ldots, CC(App_n)\}$, where $CC(App_i) = \{(CI_{i_1}, IS_{i_1}), \ldots, (CI_{i_m}, IS_{i_m})\}$ is the finite domain of App_i, namely the finite set of possible context contracts (CC) for App_i where CI are the context influences and IS is the interference specification of the contract. Furthermore, let $C = (t, R)$ be the single constraint with $t = (App_1, \ldots, App_n)$ and $R = \bigcup_{i=1}^{n} CI_{i_j} \cup CTX_{nat}(\bigcup_{i=1}^{n} IS_{i_j}) \models 0$. Thus, a solution to the problem of computing an interference resolution plan is a selection of a context contract for each application, such that the union of the context influences of all applications in combination with the natural context (CTX_{nat}) does not satisfy the union of all interference specifications.

A set of tree-based algorithms are used to solve the CSP. Each of these algorithms manages a set of context contracts for applications $App_0, ..., App_i$ that is gradually extended with contracts for applications $App_{i+1}, ..., App_n$, until a *consistent* combination of contracts – consistency in terms of a CSP equals an interference-free state in the system – is found. For this, the context contracts are *activated* at the coordinator, the context data is adjusted according to the context influences in those contracts, and the interference detection process is triggered. However, the actual context is not affected – i.e., no adaptation instructions are issued – and the active functional configurations of the applications do not change, until a plan is found and the applications are instructed to adapt.

At the beginning, the set only contains the contract of the first application. After each extension, the combination is checked for interferences and, if none are found, is extended once more, until all applications have a contract assignment. Thus, there are so called partial solution at each step, and a full solution after extending the combination of contracts for each application.

Next, the basic *informed search* algorithm is described, before it is modified in order to find the optimal solution.

5.5.2.2 Informed Search

The basic algorithm for finding a resolution plan as fast as possible follows the *explicit forward checking with conflict-directed backjumping* (FC-CBJ) strategy [86]. For traversing through the search tree, it uses the two basic functions *label* and *unlabel*. The label function is a forward step that tries to extend the existing consistent partial solution with a contract from the domain of the next application. As in the adaptation alternative search algorithms, there are the two sets *domain* and *current domain*. However, here the domain at each level is the entire set of contracts for an application and remains unchanged, whereas the current domain is a subset that only contains contracts that are consistent with the consistent partial solution. The unlabel function is a backward step. Backward steps are executed if there is no further extension possible for a partial solution. Hence, changes in the preceding part of the combination of contracts are retracted until the next consistent partial solution is found. After a successful label step on the last application, the algorithm found an interference-free solution.

While extending the partial solutions, it is possible to gather information about the relation between the contracts of the various domains. Based on this information, different forward- and backward stepping strategies are possible in order to decrease the number of steps necessary to find a solution. Of these strategies, the most informed backtracker named *conflict-directed backjumping* (CBJ), as well as the look ahead strategy *explicit forward checking* (FC) are used, as they are shown to have the best performance [86].

FC uses a special labeling algorithm in order to decrease the number of steps necessary to find a solution. While iterating through the contracts of $CC_{current}(App_i)$ during the label step for App_i, it already checks the consistency between the current contract (CI_{i_k}, IS_{i_k}) and all contracts of the succeeding applications $App_{i+1}, ..., App_n$. In case such a consistency check fails, for example between (CI_{i_k}, IS_{i_k}) of App_i and (CI_{j_l}, IS_{j_l}) of App_j, FC removes contract (CI_{j_l}, IS_{j_l}) from $CC_{current}(App_j)$. As a result, the current domains of the future applications become smaller and the algorithm has to check less contracts moving forward. If the current domain for any application in $App_{i+1}, ..., App_n$ results in being empty, no solution is possible that includes contract (CI_{i_k}, IS_{i_k}). In this case, FC reverts all changes to $CC_{current}(App_{i+1}), ..., CC_{current}(App_n)$ that were caused by labeling App_i with (CI_{i_k}, IS_{i_k}), and (CI_{i_k}, IS_{i_k}) is removed from $CC_{current}(App_i)$. As soon as FC finds a contract in $CC_{current}(App_i)$ that is consistent with at least one contract from each future domain $CC_{current}(App_{i+1}), ...,$ $CC_{current}(App_n)$, it proceeds with labeling App_{i+1}. In case $CC_{current}(App_i)$ becomes empty during labeling, the algorithm unlabels App_i.

CBJ, on the other hand, acquires information along the way about the consistency between the contracts in $CC(App_i)$ and those part of the partial solution consisting of $App_0, ..., App_{i-1}$. This information is stored for each application in so-called *conflict sets*. In case no consistent setting could be found for the partial solution and App_i, CBJ unlabels applications $App_i, ..., App_h$ where $h < i$ and h is the deepest variable in the conflict set of App_i. Hence, the algorithm jumps over all the combinations possible from $App_{h+1}, ..., App_{i-1}$ that standard *backtracking* checks, even though their involvement can not lead to a solution, as the inconsistency is caused by App_h and App_i. Further, CBJ carries the conflict set of App_i upwards to App_h during unlabeling. This way, CBJ is able to jump backwards multiple times, if necessary. In contrast, *backjumping* is only able to jump back once and then defers to standard backtracking. During unlabeling, CBJ restores the conflict sets and current domains of the applications it jumped over.

Finally, FC-CBJ combines the forward move approach of FC with the informed backtracking of CBJ. Hence, it iterates through smaller current domains during labeling, and jumps over more applications while unlabeling. The algorithm terminates as soon as it finds a solution or $CC_{current}(App_0)$ becomes empty.

5.5.2.3 Optimizing Search

The adaptation alternatives determined by the configuration management component have cost-utility-ratios reflecting their utility for the user. These individual ratings are used to optimize the individual adaptation decision. However, as previously discussed, isolated adaptation decisions can cause interferences in a multi-user system. Hence, optimizing the adaptation decision must be considered during the coordination phase as well. Representatively, in this work, optimizing means maximizing the global utility of the system, although more complex approaches, for example, involving user hierarchy or any type of credits are possible as well. Subsequently, the CSP becomes a *constraint optimization problem* and the goal is to find a solution with maximum utility. In order to be able to optimize the solution, a field for a utility value – in this case the cost-utility ratio – was added to the context contract object.

In order to find the interference resolution plan that leads to the maximum global utility, the informed search algorithm described above was modified following the *branch and bound* approach. That is, rather than terminating after finding a solution, the modified algorithm continues its search until all solutions have been found, or all potential solutions have been discarded based on their utility. Further, they follow a *breadth-first search* (BFS) approach. During labeling, the algorithms iterate through the current domain based on the utility of the contracts in descending order. Finally, the algorithms keep track of the global utility of each (partial) solution – which is the sum of the utility of all active context contracts – and backtracks as soon as the known maximum can not be outdone. That is, after successfully labeling App_i and calculating the utility of the partial solution including applications $App_0, ..., App_i$, the algorithms estimate the final global utility using the closest lower bound $|App_{i+1}, ..., App_n| * max(\bigcup_{k=i+1}^{n} CC_{current}(App_k))$. If the estimate is smaller than the best known value, the algorithm discards that partial solution and backtracks. As an example, assume the algorithm has labeled the first three of five applications with utilities of 0.6, 0.7, and 0.5, respectively, accumulating to a global utility of this partial solution of 1.8. As there are two unlabeled applications left, and the maximal possible utility left in their respective current domains is 0.8, 1.6 is added to the global utility of the partial solution, and the closest lower bound heuristic is 3.4. If the global utility of the best known

Algorithm 4 Optimal FC-CBJ-based Interference Resolution

```
 1: procedure O-FC-CBJ(n, status)
 2:     consistent ← true
 3:     status ← "unknown"
 4:     maxUtility ← 0
 5:     solution ← null
 6:     btFlag ← false
 7:     i ← 1
 8:     sortCurrentDomainsByUtilityDSC()
 9:     while status = "unknown" do
10:         if consistent then
11:             i ← o-fc-cbj-label (i, consistent, btFlag)
12:         else
13:             i ← o-fc-cbj-unlabel (i, consistent, btFlag)
14:         end if
15:         if i > n then
16:             if currentUtility > maxUtility then
17:                 maxUtility ← currentUtility
18:                 solution ← getHeadsOfCurrentDomains()
19:             end if
20:             btFlag ← true
21:             i ← i - 1
22:             deactivateCC(currentDomains[i].head)
23:             currentDomains[i].removeHead()
24:             consistent ← currentDomains[i] ≠ null
25:         else if i = 0 then
26:             if solution ≠ null then
27:                 status ← "optimal solution"
28:             else
29:                 status ← "impossible"
30:             end if
31:         end if
32:     end while
33:     return solution
34: end procedure
```

solution up to that point is lower than 3.4, the algorithm continues with labeling the fourth application. If it is higher or equal, the algorithm will not be able to surpass it by labeling the remaining two applications and can therefore discard the current partial solution.

Algorithm 4 shows the optimizing search algorithm called O-FC-CBJ. In it, three variables, namely *maxUtility* that stores the maximal known utility, *solution* that stores the solution with the maximal known utility, as well as *btFlag* were added. The flag is necessary in order to indicate whether the *unlabel* procedure should follow its respective backtracking approach, i.e., conflict-directed

backjumping, or simply backtrack one level in case the unlabel is called as part of the optimization. The details of the flag are discussed momentarily. Further, the BFS approach, i.e., choosing the domain with the highest utility first, is realized by sorting the current domain list in descending order and always retrieving the head element. This way, the algorithm needs $n * O(m \log m)$, where $m = max(|CC_{current}(App_i)|)$ once for ordering the domains, instead of $O(|CC_{current}(App_i)|)$ at each labeling. This shortcut is possible due to the static nature of the utility values of the predetermined adaptation alternatives. With dynamic values, a traditional BFS approach at each labeling would be required. Finally, the algorithm does not terminate as soon as a solution has been found, i.e., $consistent = true$ and $i > n$. Instead, it updates *maxUtility* and manipulates itself to continue its search. That is, the algorithm deactivates the current contract of the last application at the coordinator, removes it from the current domain list, and decrements the level i. Subsequently, the search proceeds in one of two scenarios: (i) The current domain of the last application is not empty. In this case, it proceeds as if it had just successfully labeled the next to last application, by labeling the last application again. (ii) The current domain of the last application is empty. Here, the *btFlag* comes into play. In this case, an unlabeling is necessary. However, the algorithm should not unlabel as with a conflict, i.e., use the conflict set to determine where to proceed, as this could lead to jumping over potential solutions. Instead, the algorithm should find the next node in the search tree. This can be achieved by proceeding with standard backtracking until an application was successfully labeled again, which then is indicated by deactivating the *btFlag*.

Algorithm 5 shows the O-FC-CBJ label function. While the current domain is not empty and the algorithm has not found a consistent state yet, it checks forward and manages the respective conflict sets. In case the algorithm finds a consistent extension, it estimates the partial solution's maximal utility and only proceeds if that estimation is greater than the known maximal utility. If the estimation – which is always an overestimate – is less than the maximal known utility, it is necessary to backtrack one level instead of following the CBJ approach, as described above. That is, in case of O-FC-CBJ, for example, the algorithm unlabels App_{i-1} instead of the maximum level in the conflict set of App_i (see Algorithm 6). Again, this is done by maintaining the *btFlag* that indicates,

Algorithm 5 Optimal FC-CBJ Label

```
 1: procedure O-FC-CBJ-LABEL(i, consistent, btFlag)
 2:     consistent ← false
 3:     while currentDomains[i] ≠ ∅ ∧ ¬consistent do
 4:         consistent ← true
 5:         activateCC(currentDomains[i].head)
 6:         for j = i + 1 → n do
 7:             consistent ← checkForward(i,j)
 8:         end for
 9:         if not consistent then
10:             deactivateCC(currentDomains[i].head)
11:             currentDomains[i].removeHead()
12:             undoReductions(i)
13:             conf-set[i] ← union(conf-set[i], past-fc[j-1])
14:         end if
15:     end while
16:     if consistent then
17:         btFlag ← false
18:         if currentUtility + estimate ≥ maxUtility then
19:             return i+1
20:         else
21:             consistent ← false
22:             conf-set[i] ← i-1
23:             return i
24:         end if
25:     end if
26:     return i
27: end procedure
```

whether the next unlabel is due to a non-consistent setting, or a discard based on the utility estimation of that setting. Hence, in case of a successful labeling, the procedure always sets *btFlag* to *false* in Line 17.

Depending on the state the *btFlag*, the unlabel procedure (see Algorithm 6) either jumps backwards following the CBJ strategy, or conducts a standard backtracking step. That is, level h is either the maximum level from the conflict set of App_i, or simply one level above i. In the latter case, the CBJ-specific *for*-loop is not executed, as $h + 1 = i$, resulting in a standard backtracking.

Both tree-based interference resolution algorithms, i.e., the *informed* and the *optimizing search*, are evaluated in detail in Section 7.1.2. Next, the application requirements are mapped to context contracts in order to use COMITY's adaptation coordination approach in this work.

Algorithm 6 Optimal FC-CBJ Unlabel

```
 1: procedure O-FC-CBJ-UNLABEL(i, consistent, btFlag)
 2:     if btFlag then
 3:         h ← i-1
 4:     else
 5:         h ← max(max-list(conf-set[i],max-list(past-fc[i]))
 6:     end if
 7:     conf-set[h] ← remove(h,union(conf-set[h],union(conf-set[i],past-fc[i])))
 8:     for j = h + 1 → i do
 9:         deactivateCC(currentDomains[j].head)
10:         conf-set[j] ← 0
11:         undoReductions(j)
12:         restoreCurrentDomains(j)
13:     end for
14:     undoReductions(h)
15:     currentDomains[h].removeHead()
16:     consistent ← currentDomains[h] ≠ null
17:     return h
18: end procedure
```

5.5.3 Application Requirements to Context Contracts Mapping

As described in Section 5.5.1.2, COMITY operates on so-called *context contracts*. For each application and functional configuration, a contract specifies the application's respective *context influences*, and which context states create an interference for that configuration of the application – the so-called *interference specification*.

As is the case with the application requirements, the COMITY framework receives information on context influences and interference specifications from the applications themselves. In contrast, however, the literals in the requirements may define multiple states, and the actual configurations are calculated at runtime. As a result, the relationship between one term of the application requirements and possible configurations is 1:n, whereas COMITY's contracts and configurations have a 1:1 relationship. Adding contracts to this framework's interface for the applications to declare would take away the possibility to use multi-state literals in the requirements term, resulting in an exponential increase in the number of terms. However, assuming global knowledge of the possible context states, and that the execution of an application can not be interfered with by a context type that is not in the application's requirements definition, it is possible to derive the context contracts automatically.

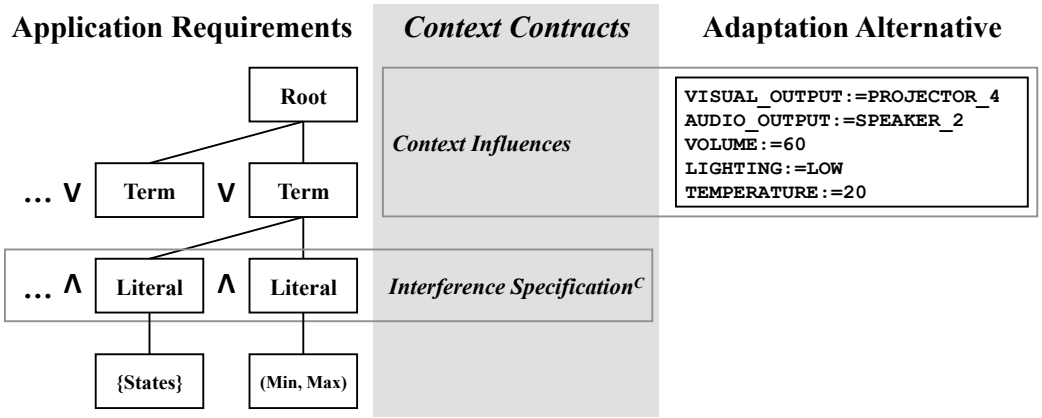


Figure 5.13: Scheme for Deriving Context Contracts from Application Requirements and Adaptation Alternatives

Figure 5.13 depicts the application requirements modeling on the left, as well as the mapping scheme from application requirements and the respective adaptation alternative to the context contracts that are needed for COMITY's interference detection. The context influences of the derived contract are simply the set of adaptation instructions of the adaptation alternative. The interference specification, on the other hand, is constructed from the application's requirements as the set of the absolute complements of the specified contexts in each term. That is, if A is the set of context states in the literals of one term, and universe U is the set of all possible context states of the variables in the term, then $U \setminus A$ is the interference specification of that application requirement term.

Following the algorithms for finding all possible adaptation alternatives of a single application in Section 5.4, and the algorithms for coordinating the adaptations of multiple applications in this section, the next section finally presents the adaptation control loop that integrates the individual components.

5.6 Adaptation Control

The previous sections of this chapter discussed the individual components that are necessary for supporting proactive adaptation. This section, finally, presents the adaptation control loop, which combines the components' functionalities into the adaptation lifecycles of the applications in the environment. The section starts by giving an overview of the entire adaptation process. Afterwards, the

individual steps are discussed in more detail. Throughout the section, the simple terms *future location* and *future context* are used for the output of location and context prediction algorithms, without suggesting certainty of the predictions.

5.6.1 Adaptation Control Process

The adaptation control loop is compiled by the individual components presented in the previous sections. Figure 5.14 shows an overview of the adaptation control process of one loop cycle. The adaptation control lifecycle for an application starts when it registers at the *Configuration Management* (CM) component and specifies its requirements (1). Subsequently, the CM subscribes to all relevant context information, as well as predictions on their respective future changes, based on the application's requirements (2). For each unique prediction request – defined by the specified context and a set of prediction tasks – the *Context Broker* (CB) selects the most suitable prediction approach from a pool, and creates and trains a new predictor implementing that approach. With any change to the knowledge base, the corresponding forecasting algorithms learn and update their predictions (3). In case of updates to subscribed current or predicted context information, the CB sends notifications to the CM (4). The CM's event handler checks the update type, i.e., *future* or *current* location/context, and initiates the subsequent procedures. In case of a prediction event, the handler triggers the pre-calculation of adaptation alternatives for the application associated to the update (5). The core task of the CM is to calculate all possible adaptation alternatives of an application based on the predicted context, i.e., proactively, as well as the current context, i.e., reactively (6). The calculated adaptation alternatives are passed to the *Adaptation Coordination* (AC) component (7), which subsequently computes interference-free adaptation plans (8) and adds them to the plan base (9). Interference-free adaptation plans are sets of adaptation alternatives – one per application – such that no context influence in the plan interferes with any requirement of another application. In case of a current information event, the handler triggers the adaptation procedures of the AC (10). If there is a pre-calculated adaptation plan for the current situation in the plan base, the AC instructs the set of necessary adaptations (11). If no pre-calculated plan exists, for example because the current situation was not predicted correctly, the AC triggers a second cycle through the adaptation control loop (12), i.e., the system

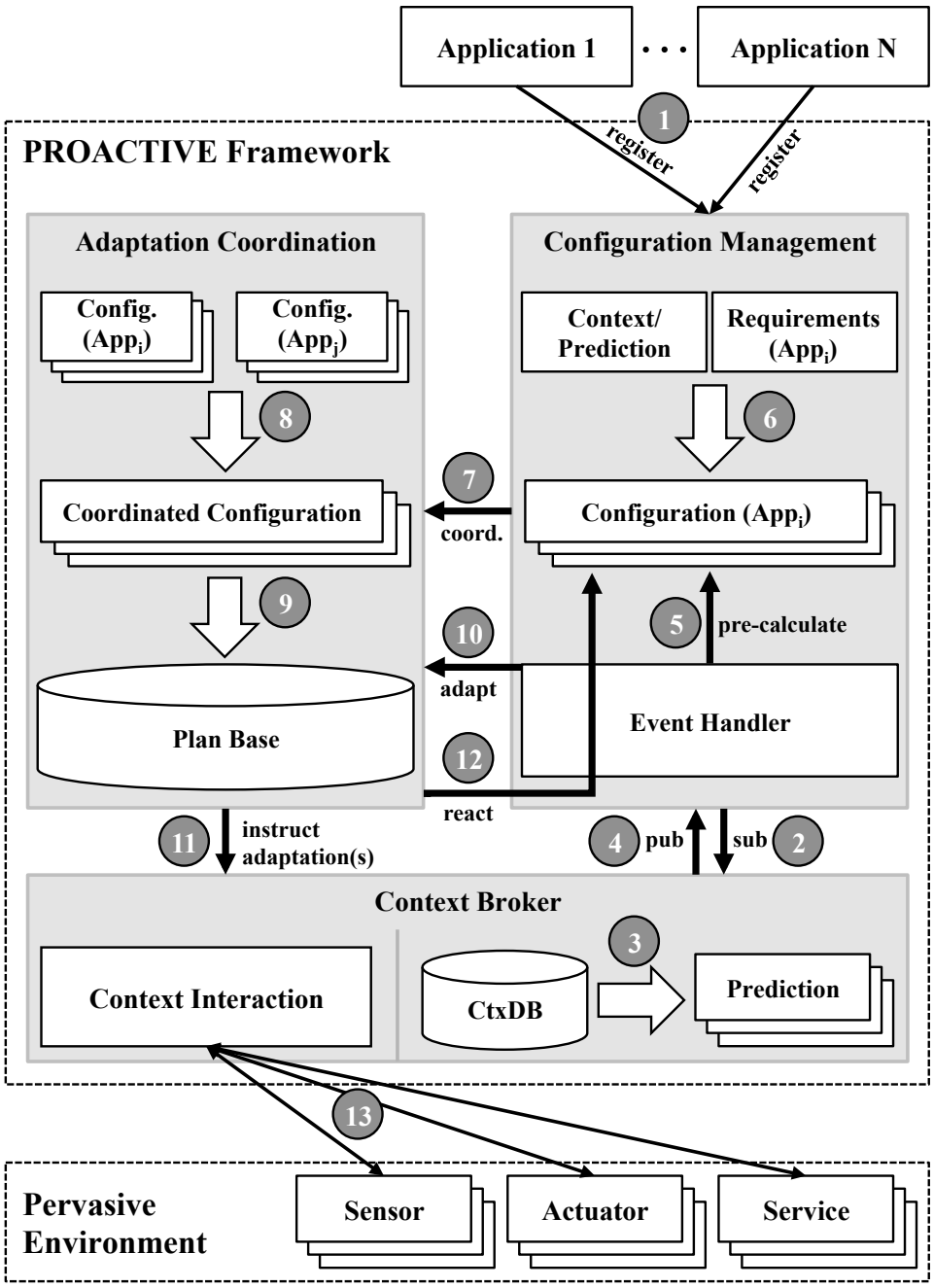


Figure 5.14: Overview: Proactive Adaptation Control Process

falls back to the reactive approach. Finally, the adaptation instructions are relayed through the context broker to the adapting entity, i.e., actuators, services, and applications (13). As discussed in Section 4.1, all entities in the system are assumed to be cooperative and execute the given instructions.

5.6.2 Context and Prediction Monitoring

Location is one of the natural indexes of context. Except for resources or services that are location-independent, an application is only concerned with the local circumstances. The framework makes use of this by structuring the context and prediction monitors along the respective application's current and future locations. That is, for each application, the CM component first extracts the list of all context variables contained in the application's set of requirements. Second, the CM subscribes to updates regarding the application's actual location, as well as its most likely next locations. Finally, for each resulting location, the CM subscribes to information on the current and predicted states of the set of context variables.

As a result, the CM monitors six different types of context events, namely `C_LOC` and `F_LOC` for the current and future locations, as well as their combinations with current and future context `C_CTX_C_LOC`, `F_CTX_C_LOC`, `C_CTX_F_LOC`, and `F_CTX_F_LOC`. Table 5.2 shows a brief overview and summary. With these monitors, the system can react to the actual situation, as well as pre-calculate adaptation alternatives for future situations, whether at the current location or at future ones. The next section describes the specific handling of each of the six update events.

	Location		Context	
Event Type	Current	Future	Current	Future
`C_LOC`	+			
`F_LOC`		+		
`C_CTX_C_LOC`	+		+	
`F_CTX_C_LOC`	+			+
`C_CTX_F_LOC`		+	+	
`F_CTX_F_LOC`		+		+

Table 5.2: Overview and Summary of Context Event Types

Algorithm 7 C_LOC Event Handling

```
 1: procedure HANDLEC_LOC(SubscriptionUpdate su)
 2:     appID ← su.getAppID()
 3:     location ← su.getLocation()
 4:     current ← true
 5:     updateMonitorsForLocation(location, current)
 6:     oldLocation ← locationRegistry.getLocation(appID)
 7:     if oldLocation ≠ NULL then
 8:         optimization ← true
 9:         triggerAdaptationAtLocation(oldLocation, optimization)
10:     end if
11:     locationRegistry.updateLocation(appID, location)
12:     updateTracker.reset(appID)
13: end procedure
```

5.6.3 Event-based Adaptation Control

Depending on the type of information update the CM receives, it triggers the appropriate handling procedure. Each procedure gets a `SubscriptionUpdate` object, that includes the monitor's ID, the ID of the associated application, and, finally, a `ContextInformation` object containing the actual update. Next, the respective handling procedure are described in detail.

5.6.3.1 `C_LOC` Event Handling

In case of a `C_LOC` event, the associated application has just changed its location. This transition makes the monitors regarding the former location obsolete. Hence, all monitors regarding the application's current location are updated first, i.e., the `C_CTX_C_LOC` and `F_CTX_C_LOC` monitors. Second, the event handler tries to optimize the global configuration at the application's former location, as the application may have restricted configurations of other applications with higher utilities. Additionally, the CM's internal location registry that keeps track of the current location of all registered applications is updated, and the CM's application-specific update tracker is reset. The update trackers are used for information bundling in the `C_CTX_C_LOC` procedure. Even though the application may need to adapt to the new location, and the applications at the new location with it, no further adaptations are triggered here. If necessary, these will be initiated from the `C_CTX_C_LOC` procedure. Algorithm 7 shows the `C_LOC` event handling in pseudocode.

Algorithm 8 F_LOC Event Handling

```
1: procedure HANDLEF_LOC(SubscriptionUpdate su)
2:     Locations ← su.getLocations()
3:     current ← false
4:     while Locations.hasNext() do
5:         location ← Locations.next()
6:         updateMonitorsForLocation(location, current)
7:     end while
8: end procedure
```

5.6.3.2 `F_LOC` Event Handling

With each change of the current location of an application, the knowledge base of the location predictors for that application also change. The predictors learn of the change and recalculate their predictions. In most cases, this will result in a different set of expected location transitions and subsequently create `F_LOC` events. As with the `C_LOC` event, the event handler mainly updates the respective monitors. However, depending on the prediction task properties, the update may contain a sequence of next locations, or a set of next location predictions that, for example, accumulate to a confidence rate of 90%. Hence, the handler needs to update the monitors for each of the predicted locations (see Algorithm 8). Again, the necessary further steps, like pre-calculating adaptation alternatives, will be triggered by the respective procedures that receive context state updates.

5.6.3.3 `C_CTX_C_LOC` Event Handling

The `C_CTX_C_LOC` events are the most complex to handle in that the procedure can be triggered in various cases. First, it can be the consequence of changes to the application's environment by either natural effects, e.g., the setting sun, or by other applications joining the environment. Second, it can be due to the transition to a different runtime environment after a preceding location change. For both cases and for each event, the event handler first updates the *situation* of the application (see Section 5.6.4). Afterwards, the handler checks the necessity of adapting, as the new context may still satisfy the active requirements term of the application. If it does not satisfy the term, an adaptation is indispensable. If it does, an adaptation is not necessary per se, but there may be potential for optimizing the system's overall configuration. Hence, the handler triggers the

Algorithm 9 C_CTX_C_LOC Event Handling

```
 1: procedure HANDLEC_CTX_C_LOC(SubscriptionUpdate su)
 2:     appID ← su.getAppID()
 3:     location ← su.getLocation()
 4:     if location ≠ locationRegistry.getLocation(appID) then
 5:         return
 6:     end if
 7:     type ← su.getVariableType()
 8:     state ← su.getVariableState()
 9:     current ← true
10:     updateSituOfApp(appID, location, type, state, current)
11:     updateTracker.addUpdate(appID, type)
12:     Variables ← ApplicationRegistry.getDistinctVariables(appID)
13:     if ¬updateTracker.containsAll(appID, Variables) then
14:         return
15:     end if
16:     optimization ← false
17:     activeTerm ← ApplicationRegistry.getActiveTerm(appID)
18:     if SituationManagement.satisfies(location, activeTerm) then
19:         optimization ← true
20:     end if
21:     triggerAdaptationAtLocation(location, optimization)
22: end procedure
```

adaptation process regardless, indicating the necessity using the `optimization` flag. The effects of the flag on the adaptation process are described in Section 5.6.5. In the second case only, i.e., `C_CTX_C_LOC` events subsequent to a location change, the handler should wait for complete information about the new situation before taking action. For this, the CM tracks the updates for each application at its current location. The information on the new situation is complete as soon as the CM receives an update for each context variable in the requirements of the application, and the event handler can proceed with its procedure.

Finally, debugging has shown that the CM might receive outdated context updates, especially if the simulation is accelerated. This is due to the asynchronous communication in the system. However, it is only significant for `C_CTX_C_LOC` events, as they do not trigger pre-calculations, but instead actual adaptations. Hence, the handler needs to filter out such outdated messages, which it does using location filters. Algorithm 9 shows the entire event handling procedure.

Algorithm 10 F_CTX_C_LOC Event Handling

```
 1: procedure HANDLEF_CTX_C_LOC(SubscriptionUpdate su)
 2:     appID ← su.getAppID()
 3:     location ← su.getLocation()
 4:     if location ≠ locationRegistry.getLocation(appID) then
 5:         return
 6:     end if
 7:     type ← su.getVariableType()
 8:     state ← su.getVariableState()
 9:     current ← false
10:     situation ← updateSituOfApp(appID, location, type, state, current)
11:     triggerpre-calculation(appID, location, situation)
12: end procedure
```

5.6.3.4 `F_CTX_C_LOC`, `C_CTX_F_LOC`, and `F_CTX_F_LOC` Event Handling

The final three update event types each trigger the pre-calculation of adaptation alternatives (see Section 5.6.4). The data that the three respective event handling procedures receive contains both context and location information, from which at least one is a prediction.

In case of predicted context changes at the current location of the application, i.e., a `F_CTX_C_LOC` event, the received information is first relayed to the situation management component. Afterwards, the handler triggers the pre-calculation of adaptation alternatives based on the received information, which is either the predicted context at the current location, the current context at a predicted location, or the predicted context at a predicted location, respectively. For `F_CTX_C_LOC` events, the handler additionally checks for outdated updates due to delays stemming from the asynchronous communication pattern, again using location filters. This filtering is not possible for events including predicted locations, as there may be more than one valid future location the system wants to prepare for. Algorithm 10 shows the procedure for handling `F_CTX_C_LOC` events as the representative for all three update types. The other two procedures differ in that they do not filter for outdated messages (Lines 4-6). Additionally, in the procedure for handling `C_CTX_F_LOC` events, `current` gets `true` (Line 9), as the event relates to current information.

The next section describes how the component manages the pre-calculated adaptation alternatives and coordinated adaptation, and provides fast access to them.

5.6.4 Adaptation Alternative and Plan Management

The event handling procedures presented in the previous section trigger the configuration algorithms of the framework whenever adaptations have to be (pre-)calculated. As presented in Section 5.4, these algorithms find all possible adaptation alternatives of an application – including location adaptations – given the application's requirements, the context in the environment (current or predicted), and the adaptability of the context at each location in the environment. The search for adaptation alternatives is modeled as a CSP, and various backtracking-based algorithms find all solutions through a complete search process. Subsequently, each adaptation alternative gets a utility value assigned to it based on either user preferences or duration-dependent utility and costs metrics. As a result, the framework has to handle a huge set of application and situation-specific adaptation options.

In order to manage these calculated adaptation alternatives efficiently, i.e., storing them in a data structure with random access for fast retrieval, the adaptation control does not only use the application ID and location as indexes, but also the concept of *situations*. A situation is the combination of the set of context variables and their respective active state at a location, e.g., `VISUAL_OUTPUT:=IDLE, AUDIO_OUTPUT:=BUSY` at `AUDITORIUM_1`. The set of context variables included in the specific situation depends on the requirements specification of the individual application. That is, if an application is only dependent on the state of two of several context variables, only those two define the situation for the application. Each situation is identified by the hash value of its context combination. To make these *situation ID*s comparable and reproducible so that they are suitable as an index of the adaptation alternatives data structure, the variable-state-pairs are comparable and organized in a *tree set*, which guarantees ordering. With the situation ID as an additional index, the adaptation control can retrieve all adaptation alternatives of an application at a given location with the given conditions in constant time ($O(1)$), or check whether there are adaptation alternatives, respectively. Further, the tree set structure of the `Situation` objects allows to check whether the situation in question satisfies a specified requirements term of an application in $O(|var|)$, where $|var|$ is the number of context variables the application relies on.

All current and projected situations for each application at each location are administered by the *Situation Management* (SM) component. That is, the component contains two registries, one for the current situation at each location in the environment, and one for the future situation at each location. The registries track the actual situation of each application, as well as which situation is expected either through a change of the context at the location of the application, or through a location change of the application itself. The SM receives all information and prediction updates from the event handler's respective procedures (see Section 5.6.3).

Adaptation in multi-user environments additionally requires coordination, as isolated adaptations of single applications that share context often leads to oscillating effects. For this, Section 5.5 showed how to integrate the COMITY framework for adaptation coordination, i.e., derive COMITY's *context contracts* from the application requirements in order to use COMITY's interference detection. Further, it presented a new set of tree-based interference resolution algorithms that are able to optimize the global utility in the environment. After each calculation of adaptation alternatives, the adaptation control automatically triggers the coordination algorithms that compute all interference-free *adaptation plans* for the location for which the adaptation alternatives were calculated. Even though one interference-free adaptation plan would ensure the execution of the applications, the goal is to optimize the environment's global utility. Hence, the framework has to test all combinations of adaptation alternatives for interferences and subsequently resolve them. This includes all the different combinations of applications, as the set of pre-computed adaptation plans should include those for the case that the set of applications at the location suddenly changes. Accordingly, the process of computing coordinated adaptation plans is as follows. First, the framework (pre-)calculates all adaptation alternatives of an application for the given context/prediction at location l, and adds them to the plan base. Second, it takes the set S of applications currently at l, as well as those predicted to join l, and form the set S' of all nonempty subsets of S, i.e., the power set of S excluding the empty set. For each set S'_i in S', the framework then loads the context contracts of all adaptation alternatives for location l of all applications in S'_i into COMITY's coordination component. Then, the framework triggers the resolution algorithms for maximizing the global utility of the system, and

receives a set of adaptation plans – including the optimal one – for the set of applications in S'_i at location l. After doing this for all sets in S', and adding the solutions to the plan base each time, the adaptation control either terminates the pre-calculation process, or proceeds to the adaptation process in case this cycle is part of a fallback reactive adaptation.

The plan base itself contains a registry that stores the coordinated adaptation plans with random access. For this, it uses the location identifier and the respective set of application identifiers, for which the adaptation plan applies, as indexes. As with the situation ID, the *application combination ID* – as the hash value of a sorted list – is comparable and reproducible.

The next section discusses the adaptation process, i.e., when and how to fetch and distribute adaptation instructions, before Section 5.6.6 presents the architecture of the adaptation control component.

5.6.5 Adaptation Process

Adaptation in the system occurs following two types of update events. First, a `C_CTX_C_LOC` event results in a situation, in which at least one application at that location can not continue to provide its service. In this case, an adaptation is indispensable. Further, a joining application that is still able to use its instantiated functional configuration, as well as a context change due to natural effects instead of actuation, may result in a functional but non-optimal overall configuration. These two optimization situations are recognized while handling `C_CTX_C_LOC` events as well. Second, an application has transitioned to a different location, as indicated by a `C_LOC` event. This also potentially creates room for optimizing the total utility, albeit of the remaining applications at the previous location.

As discussed in the previous section, adaptation in multi-user systems must be coordinated and target all applications at a location. Hence, the adaptation process starts by getting the set of application identifiers that are currently present at the location in question, respectively the application combination ID of that set. With that ID, the framework can then query the plan base to see, whether a pre-computed adaptation plan exists or not. If so, it chooses the plan with the highest total utility and proceeds to instruct the set of adaptations. If no

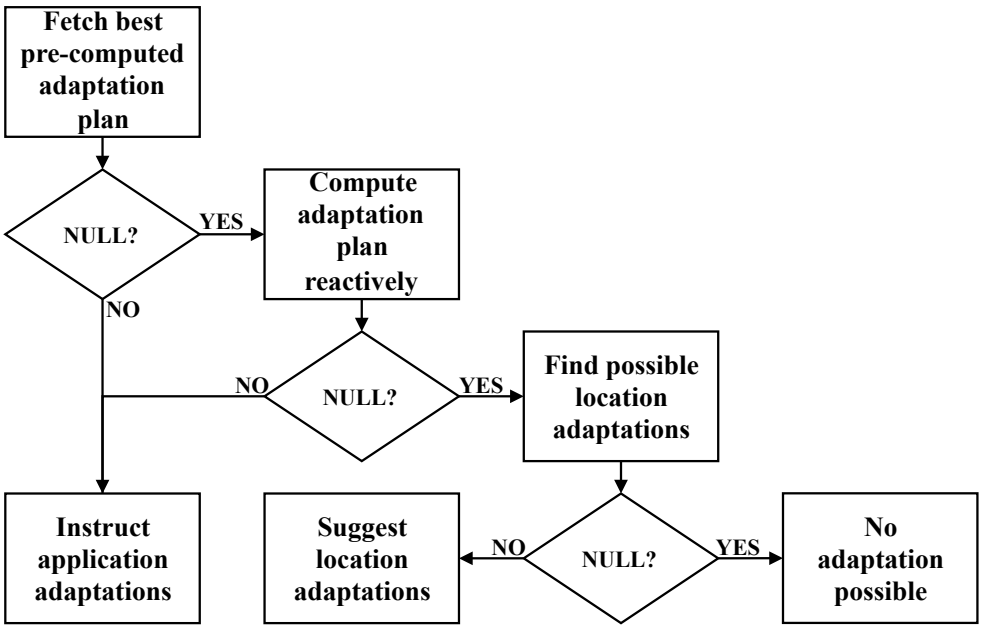


Figure 5.15: Flow Chart of the Adaptation Process

pre-computed plan exists, for example, because the current situation forcing the adaptation was not predicted correctly, the framework tries to adapt reactively. That is, if it is unknown whether there exists an interference-free adaptation plan for the set of applications in the current situation, the adaptation plan computation is triggered at this point in time. If no solution is possible, or none have been found in the reactive cycle, respectively, the framework proceeds to adapt the location of the applications, and, hence, the location of their users. As discussed in Section 4.1, *user adaptations* are merely suggestions by the framework. In this instance, suggestions of where the users' applications can provide their services. Nevertheless, the framework searches for alternative locations that can fulfill the applications' requirements. To do so, it calculates the adaptation alternatives for each location in the environment in the order of their respective distance to the current location, starting with the nearest one. The utility of all alternatives is then weighted using the respective distance, and the users are informed of his/her options. In the case that there is also no location adaptation possible, the respective application can currently not run as intended at any location in the environment. If the reactive adaptation plan computation cycle was successful, the framework again proceeds to instruct the set of adaptations. Figure 5.15 shows the overall adaptation process as a flow chart.

The instructions in the adaptation plan are executed sequentially for each application in no particular order. That is, the adaptation thread gets the set of adaptation alternatives making up the plan – one per application – and instructs them. These adaptation alternatives contain a mapping of context variable types to context variable states that represent a specific functional configuration of its application, as well as what type of adaptation the respective mapping is. An example of an adaptation alternative is as follows:

```
AA = {VISUAL_OUTPUT:=PROJECTOR_4, COM;
      AUDIO_OUTPUT:=SPEAKER_2, COM;
      VOLUME:=60dB, CTX;
      LIGHTING:=LOW, CTX;
      TEMPERATURE:=20, CTX}
```

The adaptation instructions are relayed through the environment's central context broker. The broker uses the annotated adaptation type to direct the instructions correctly. In case of behavior or composition adaptations, the broker informs the applications of how they should adapt (cf. *application-controlled automatic adaptation* in Section 4.1). In case of context adaptations, it directly instructs suitable actuators at the target location, which is passed along to the broker with the instructions.

5.6.6 Component Architecture

Figure 5.16 shows the architecture of the adaptation control component. The various types of bold lines indicate the main interaction patterns in the component, as they are described in Sections 5.6.3 and 5.6.5. Of the bold lines, the black ones show the proactive adaptation cycle with pre-calculated adaptation plans, whereas the gray lines depict the follow-up reactive cycle if there are no pre-calculated plans.

As described in Section 5.3, applications register at the framework with their requirements. Subsequently, the event handler triggers the necessary management duties, pre-calculations, and adaptations based on the event updates it receives from the context broker. The five main components of the adaptation control unit are implemented as threads and thread pools, respectively, as indicated in the figure by rotating arrows. Communication between the individual threads is realized via so-called blocking queues. This asynchronous implementation guarantees that the components do not have to wait for each other, and that no update events or calculation/adaptation tasks are dropped due to busy components. If a thread is/becomes idle, it immediately takes up its new task if existent, or as soon as a task is added to its queue, respectively.

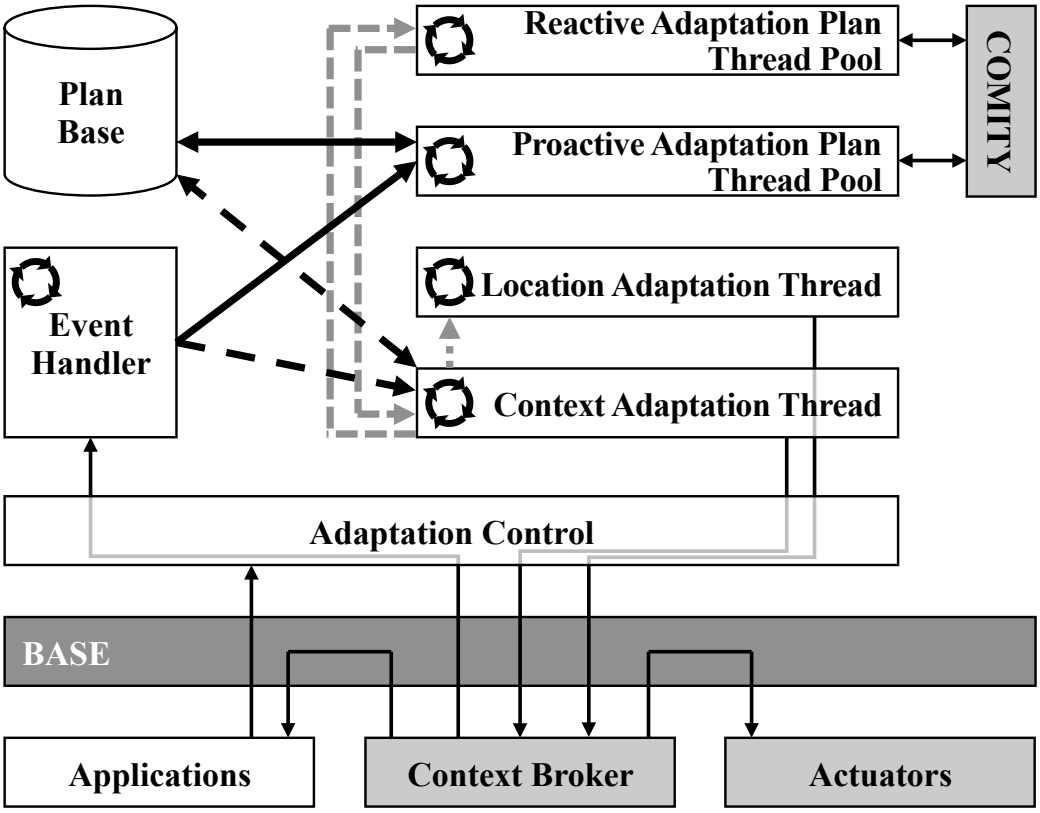


Figure 5.16: Architecture of the Adaptation Control Component

Further, the threaded implementation allows to purposefully interrupt a busy thread. This is especially important for the *Reactive Adaptation Plan Threads*. For very complex or unsolvable CSPs, finding a solution or terminating after scouring the entire search space, respectively, may take up several minutes or hours. By that time, the situation might have already passed, making the calculation obsolete. Hence, the framework interrupts a busy thread as soon as the situation at the location in question has changed. However, in case the respective calculation thread is busy, but the situation is not obsolete, it should not be interrupted. Therefore, the framework uses thread pools for the proactive and reactive adaptation plan calculation threads, with one thread per location in the environment. In the reactive cycle, the 1:1 location to thread ratios is sufficient, as with any new task for the thread, the previous one automatically becomes obsolete. For the proactive cycle, there may exist several tasks regarding the same location at the same time. The 1:1 ratio worked well in the simulations of the framework evaluation (see Section 7), but the number of threads can easily be adjusted, if necessary.

```
interface Instructable {
     void instructAdaptation(
            ContextVariableType cvt,
            ContextVariableState cvs);
     void instructAdaptation(Location l);
}
```

Figure 5.17: The interface to be implemented by all applications.

For adapting the applications and/or context, the context or location adaptation thread sends the adaptation instructions to the context broker, who distributes them to the respective actuators and applications. For this, the applications have to implement the *Instructable* interface as shown in Figure 5.17. This interface completes the framework.

5.7 Summary

This chapter presented a general framework for proactive adaptation with middleware-based system support. The framework offers uniform context interaction and allows applications to specify their context requirements as well as user preferences. With this, the framework calculates all possible adaptation alternatives of an application based on current or predicted context, and coordinates these adaptation alternatives to form adaptation plans. Finally, the framework monitors relevant context events based on application requirements, and automatically triggers necessary adaptations.

The next chapter gives implementation details of the prototype system, before Chapter 7 evaluates the framework.

6 Prototype

The previous chapter presented a general framework with system support for proactive adaptation in pervasive systems. This chapter describes implementation details of the prototype, before Chapter 7 evaluates the algorithm performance and the overall prototype in real-time simulations. First, Section 6.1 details the prototype system. Afterwards, Section 6.2 presents the prototype's architecture. Details of the context database are given in Section 6.3, before Section 6.4 presents the task-based predictor selection scheme. Finally, Section 6.5 discusses limitations of the current prototype system.

6.1 Implementation Details

All components of the prototype system are implemented as so-called *BASE services* in Java, more precisely Java Platform, Standard Edition 6 (Java SE 6 or Java SE 1.6.0, respectively [72]; Java SE 6 API [75]). Compliant with the resource-restricted nature of mobile and embedded devices, the four interfaces context service, sensor, actuator and subscriber do not rely on any Java libraries that are not included in the Java Platform, Micro Edition (Java ME [71]), more specifically Java ME's Connected Limited Device Configuration Version 1.1 (Java ME CLDC 1.1 API [74]). The centralized entities, on the other hand, are intended to run on resource-rich machines in the space and, therefore, are not subject to any library limitations. Finally, the context broker is connected to an instance of the open source database MySQL Community Server 5.1.49 [73].

In total, the prototype system (without BASE) consists of 163 classes and $11,129$ lines of code (abstract devices: $2/200$; context broker: $78/4,915$; adaptation control component including configuration and coordination algorithms: $58/4,057$; context predictors: $24/1,838$; simulation manager: $1/119$).

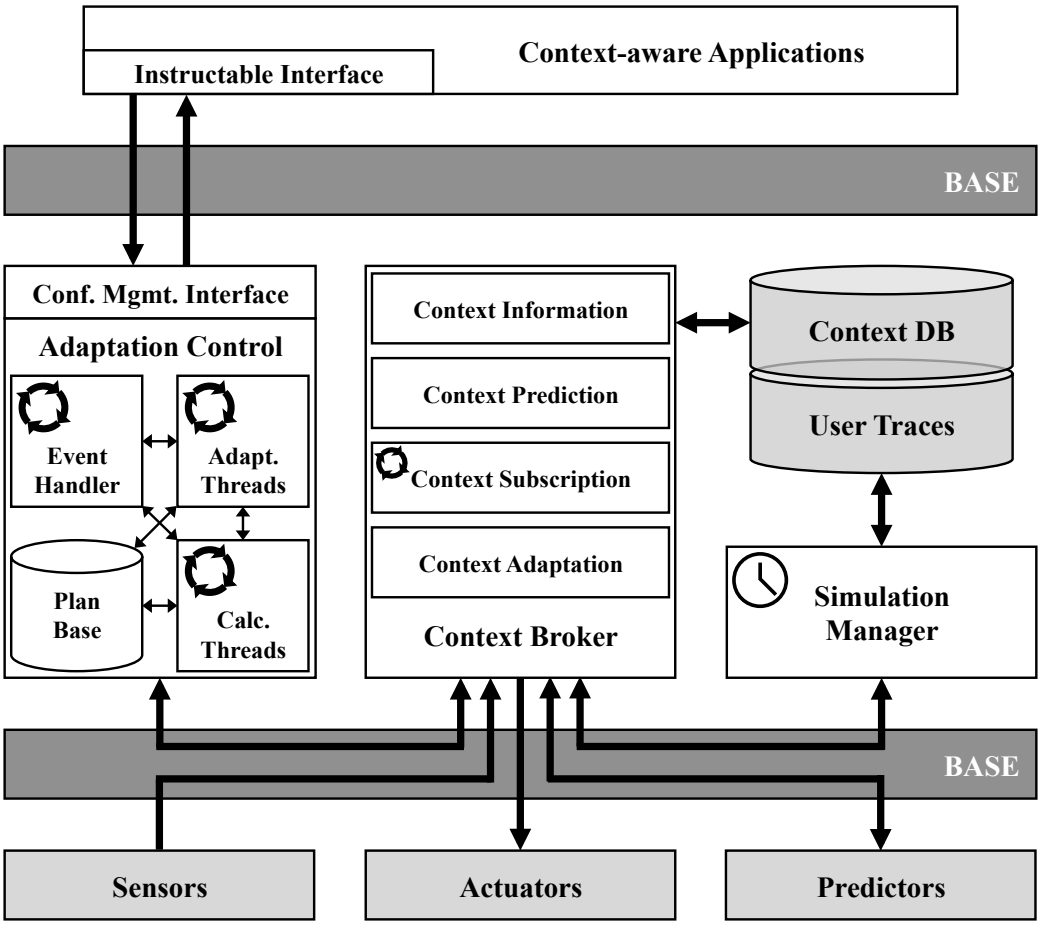


Figure 6.1: Architecture of the Framework Prototype

6.2 Prototype Architecture

The prototype system consists of the centralized context broker and adaptation control component, as well as applications, sensors, actuators, and predictors. Further, it features a simulation manager for evaluation purposes with its own database instance that primarily feeds context events into the system, e.g., real user traces, but can also accelerate the simulation time (cf. Section 7.2).

Figure 6.1 depicts the architecture of the prototype framework. The applications register with their set of requirements and preferences at the adaptation control component via the configuration management interface defined in Section 5.4.3. Subsequently, the adaptation control sets up all necessary context and prediction monitors by subscribing to the respective context query (cf. Section 5.2.2) at the context broker (cf. Sections 5.2.3 and 5.2.4), and triggers (pre-)calculations, (pre-)coordinations, as well

Table	***temperature***				
Variable	*location*	*timestamp*	*state*	*confidence*	*variance*
Data type	VARCHAR(128)	TIMESTAMP	VARCHAR(128)	DOUBLE	DOUBLE
Property	NOT NULL	NOT NULL	NOT NULL		
	PRIMARY KEY				
Example	DE:MA:L15:1-6:723	2010:11:02:12:00:00	21.5	0.9542	0.173

Table 6.1: The structure of the context database schemes.

as adaptations depending on the subscription updates (cf. Sections 5.6.3 and 5.6.5). In order to receive adaptation instructions, the applications implement the `Instructable` interface (cf. Section 5.6.6). During simulation, the simulation manager feeds records of future context information into the system as if it were a sensor, i.e., using the broker's `reportContext` method. Additionally, it offers its internal clock to all the other components via the context broker, in case simulation acceleration is activated. Finally, as described in Section 5.2, all context interaction is mediated by the context broker in order to provide uniform access.

6.3 Context Database

In order to store and retrieve context information, the prototype's context broker is connected to a MySQL database. Table 6.1 shows the structure of the framework's context database schemes by example of the context variable *temperature*. In general, the 5-tuple consists of the location information, e.g., symbolic coordinates, the timestamp of the measurement, the context variable state itself – encoded as text to cover symbolic and numeric values – as well as the confidence and variance of the information. Some sensors and predictors may not be able to provide the two information quality metrics. Hence, although not desirable, the confidence and variance values may be null, i.e., empty.

Location and timestamp together form the primary key of each relation. That is, the combination of location and time is unique in the table and distinguishes the set of attributes. For each location and point in time, whether sensor measurement or prediction, the database contains at most one set of context information per context variable. In case of multiple sensor readings with the same timestamp and location, the one with the highest quality is inserted/kept in the database. For predictions, the most recent information is stored. Their information quality metric, e.g., probability or similarity score, is not absolute, but relative to the other possible outcomes in that

respective prediction attempt, and hence not comparable. Typically in highly dynamic environments, predictions become more accurate, the closer they are to the actual event. Finally, as time progresses, predictions are automatically replaced by sensor readings.

6.4 Task-based Predictor Selection

In order to determine adaptation alternatives for future context changes, context prediction is a prerequisite. This includes forecasting the state of the pervasive environment, as well as the availability of context services. As described in Section 2.5, there exist several context prediction approaches. However, no single approach is suitable for all types of prediction tasks [107, 16]. The algorithms, for example, differ in the data types they can process, e.g., numeric vs. symbolic representation, their ability to accurately predict events in the short and longterm future, and whether they are suited for running on patchy data sets. Petzold [83] uses a combination of prediction approaches in order to support a broad range of prediction tasks. However, the task-to-approach-mapping is static. In this work, the context broker's prediction engine uses a task-based selection scheme that selects the best fit prediction approach from the set of available predictors at runtime.

For this, the engine uses the following parameters and respective values, that define a prediction task in the framework. The list of parameters is not meant to be exhaustive, but to distinguish between the set of prediction approaches implemented in the prototype.

1. **Data type:** A fundamental difference between context information is their respective data type. Most low-level context – i.e., raw sensor data, such as temperature or GPS coordinates – are expressed using *numeric* values, whereas high-level context – i.e., inferred or abstracted context – usually is encoded with *symbolic* labels. In any case, these different data types are a factor in whether a prediction method is applicable or not. Hence, the first parameter is *data type* $= \{num, sym\}$.
2. **Dimension:** Context time series may be multi-dimensional, i.e., more than one set of data is needed to describe the context. However, not all prediction approaches can operate on multi-dimensional datasets, or their runtime complexity becomes too high for real-time systems. Hence, the parameter *dimension* $= \{one, multi\}$ identifies the predictors that are suitable for multi-dimensional context.

3. **Horizon:** The horizon of a prediction describes how far it reaches into the future. However, in this instance, *far* does not imply a time interval in seconds per se. Most of the times, it refers to the amount of state changes to a context that a prediction approach can output, including only the next, i.e., one. Further, [60] finds that there is a notable drop-off in quality with some approaches after five state change predictions. Hence, the parameter is $horizon = \{next,\ short,\ long,\ date\}$, with $short \leq 5$, $long > 5$, and *date* identifying methods that can predict context at a specific point in time or in t amount of time, respectively.

4. **Dataset:** The various prediction methods are differently capable of dealing with gaps and errors in the recorded context history. For example, pattern matching approaches are more robust against gaps than transition probability approaches. The parameter $dataset = \{reliable,\ patchy\}$ allows to characterize the dataset. However, the quality of the dataset has no effect on whether an algorithm is able to compute an output or not. Hence, the parameter is optional.

5. **QoS:** Depending on their scenario and current situation, applications may have different additional requirements towards context prediction. As an example, an emergency monitoring system may accept a higher false positive rate in return for a minimal false negative rate. The complexity of a prediction algorithm has no general implication on the accuracy of its results. However, experimental results indicate that simple approaches have a very fast response time at the cost of accuracy. For some applications, a rough but quick estimate may be sufficient. On the other hand, it is possible to increase the soundness of a prediction at the cost of response time by aggregating the results of several predictors. An optional QoS parameter allows to model such additional characteristics. In the prototype, it is limited to the complexity range with $QoS = \{simple,\ complex\}$, where *simple* identifies methods with a complexity in $O(n^2)$.

The parameter values are not distinctive, i.e., a prediction approach may be applicable for multiple values. More specific, for some parameters, the values have an *include* relation. For example, a prediction approach that is suitable for predicting context with a *long* horizon also satisfies the characteristics *short* and *next*. In terms of set theory, where A^{p_i} is the set of prediction approaches satisfying parameter p_i, the relation is $A^{long} \subset A^{short} \subset A^{next}$. Similarly, methods that can handle multi-dimensional context time series can also process one-dimensional time series, i.e., $A^{multi} \subset A^{one}$, and methods that compute good results based on patchy datasets also do so on consistent ones, i.e., $A^{patchy} \subset A^{reliable}$. Additional classification of numeric data, e.g., assigning labels such as *warm* and *cold* to temperature ranges, allows the relation $A^{sym} \subset A^{num}$.

Approach	Data Type	Dimension	Horizon	Dataset*	QoS*
AA	num/sym	multi	long/date	patchy	complex
LR	num	one	short/date	patchy	simple
MM	sym	multi	short	reliable	simple
SOM	num	one	next	patchy	complex
SP	sym	one	next	reliable	simple

* = optional parameter
AA: Alignment Approach, LR: Linear Regression, MM: Markov Model, SOM: Self-Organizing Map, SP: State Predictor

Table 6.2: Overview of the prediction approaches implemented in the prototype with regard to their respective prediction task parameters.

However, this comes with information loss and is not implemented in the prototype. Finally, include relations may also be applicable to QoS parameters, but the values *simple* and *complex* featured in the prototype are distinctive.

The prototype framework features five predictor implementations based on the prediction techniques alignment approach [110], linear regression [41], Markov model [12], self-organizing map [59], and state predictor [84]. Table 6.2 summarizes the list with regard to the prediction task parameters that the individual approaches fulfill. The prediction engine selects the best fit approach from the pool of predictor services in the environment at runtime based on the specifications in the prediction task. In the prediction task, the context variable implicitly declares data type and dimension of the context, whereas the application explicitly specifies the horizon in terms of the amount of state changes, or a specific date for the prediction, respectively. From the optional parameters, the application specifies its desired QoS characteristics, whereas the prediction engine itself decides on the reliability of the dataset based on its consistency, with *patchy* being the default assessment.

6.5 Limitations

The current prototype framework has the following limitations as it pertains to the theoretical framework and the extent of the simulations. First, the utility metrics of the adaptation alternatives and the coordinated adaptation plans used in the prototype system are not duration-dependent but static, and specified in the application requirement terms instead of being calculated for each adaptation alternative using the utility and cost functions.

Second, natural effects on context, e.g., the different levels of natural light throughout the day, are not simulated. As a result, context that has been actuated to a certain state remains in that state until explicitly changed. However, such natural or any other effects in a real environment are accounted for by the context monitors in the adaptation control component. As soon as the state of a context changes, the monitors inform the event handler, which re-actuates the context to fit the current adaptation plan, if necessary.

Finally, as natural effects are not simulated, unsolvable situations can only emerge from valid configuration instantiations if a new application joins a location. Hence, the ensuing search for location adaptations is only conducted for that new application, as it causes the interference. A productive system should feature a more sophisticated approach to unsolvable situations, for example, including negotiation.

This chapter detailed the prototype implementation of the framework. The next chapter evaluates the performance of the search algorithms individually, as well as the entire framework simulations using real user traces.

7 Evaluation

The last chapter described implementation details of the prototype framework. This chapter evaluates the prototype framework on two levels. First, the adaptation alternative and adaptation plan algorithms are evaluated in isolation with regard to performance metrics such as runtime in ms and number of steps using different problem sets. Afterwards in Section 7.2, the prototype framework is applied in real-time simulations of an interactive pervasive system using recorded user traces in order to analyze its behavior, benefits, and overhead. Finally, the evaluation chapter closes with a discussion in Section 7.3.

7.1 Algorithm Performance

In this section, the two main algorithm groups are evaluated in isolation. First, Section 7.1.1 discusses the adaptation alternative search algorithm and its variations, which find all adaptations that are possible for an application in the given environment of current or predicted context. Afterwards, Section 7.1.2 examines the adaptation coordination algorithms that search for interference-free adaptation plans for multiple applications in a shared context.

7.1.1 Adaptation Alternative Algorithms

The variations of the adaptation alternative search algorithms are evaluated regarding their scalability and asymptotic behavior with growing complexity of the underlying CSP. First, a neutral test case with regard to the application requirements' complexities and available context services at each location is used to establish a baseline. Then, two additional test cases are used that are designed to isolate the features of the ordering heuristic and the context service index structure. The evaluation was conducted using a standard desktop PC with an Intel Core i5-2500K CPU (four cores with 3.3 GHz each), 8 GB of main memory, and equipped with a 64-bit Windows 8 operating system.

To evaluate the algorithms' scalability, their performance was measured for different problem sizes, as defined by the following parameters: (i) the number of lo-

cations $N = \{10, 25, 50, 100, 250, 500, 1000\}$, (ii) the number of context services per location $M = \{1, 2, 3, 4, 8\}$, (iii) the number of locations with solutions for the CSP $R = \{1, 10\ \%, 20\ \%, 30\ \%\}$ (1 is absolute; percent values refer to the total number of locations), and (iv) the least number of possible states for a context service. In order to compensate for possible variations in the system's performance, the measurement results are the average values of ten runs.

The three test cases establish a performance baseline, evaluate the benefits and overhead of the ordering heuristic as well as the context service index structure, and analyze the algorithms' behaviors with growing problem complexity are as follows:

The first test case should provide a baseline performance measurement for the basic exhaustive search algorithm, i.e., the scenario should be neutral with regard to the ordering heuristic. In it, the available context services are similar at each location in terms of their respective context variables, as well as their respective context variable states. Further, the applications' requirements are constructed to have similar complexity indexes, minimizing the effect of the heuristic. Hence, the differentiation between a valid and a non-valid location is only possible via the specific states of each individual service, forcing the algorithm to traverse almost the entire search tree.

In the second scenario, the similarity of the locations in the environment is reduced. That is, non-fitting locations have a different context service infrastructure as locations with solutions and can be disqualified by this property. For example, for an application requiring five different context services, the algorithm can directly disqualify all locations that do not feature at least these five. The complexity of the requirements themselves is similar to the setup of the first scenario.

In the third scenario, the locations have a similar context service infrastructure. All locations are fully compatible with the applications' requirements with regard to the type of context variables. However, some random services are not able to fulfill the specific context states that are demanded by the requirements. The complexity indexes of the requirements correlate to the actual service occurrences in the environment, simulating more and less frequently used context services. In this test case, the ordered search should be able to capitalize from its most constraint variable heuristic.

Figure 7.1 depicts the runtimes in *ms* from the first test case that establishes the baseline for each algorithm variation, i.e., the exhaustive (E-DFS) and the ordered search (O-DFS), both with and without the index structure's caching of available context services (w/ and w/o CSIS). The results show that the ordered search using the index structure performs best in each situation. Hence, the trade-off between its over-

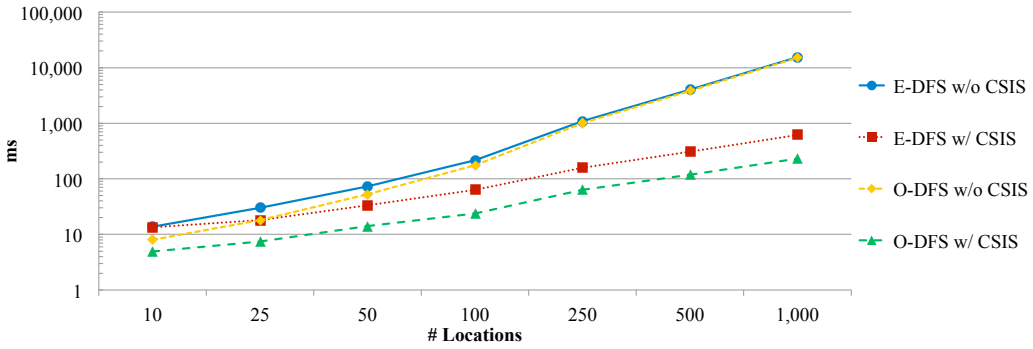


Figure 7.1: The results of Test Case 1 for the exhaustive search (E-DFS) and the ordered search (O-DFS), with and without the context service index structure (w/ and w/o CSIS), and $m = 4$.

head and benefits from the combination of the constantly maintained index structure and the complexity heuristic is more than positive. In contrast, the exhaustive search without index structure is the slowest algorithm, irrespective of the setup or the number of locations.

The biggest difference in these baseline performance measurements is between the algorithm variations using the index structure, and those not using it. Without the index structure, the algorithm is not able to filter out locations that automatically disqualify for being part of possible solutions based on their context service infrastructure (cf. Algorithm 1, Lines 8-12). Instead, the algorithm tries to find solutions for those locations anyway, terminating after the first variable in the requirements can not be labeled. Hence, the runtimes of both E-DFS w/o CSIS and O-DFS w/o CSIS grow at a far greater pace than their counterparts E-DFS w/ CSIS and O-DFS w/ CSIS, respectively. Case in point, for 500 locations, O-DFS w/ CSIS is the fastest with 119 ms, while E-DFS w/o CSIS is the slowest with $4,042$ ms. (The average of all variations is $2,079$ ms.) For $1,000$ locations, O-DFS w/ CSIS takes 230 ms, while E-DFS w/o CSIS needs $15,520$ ms, constituting growth rates of 194 and 384 %, respectively.

Further, it is notable that even though the context service infrastructure is very similar at each location, the ordering heuristic already provides a benefit. Initially, O-DFS w/o CSIS actually performs better than E-DFS w/ CSIS, due to the heuristic's benefit. However, this changes starting at approximately 25 locations in the environment, at which point the benefits of the prior filtering of the search space starts showing its effect, by outweighing the associated maintenance overhead. In general, the ordering heuristic has a small effect in Test Case 1, as the two variations with IS, as well as the two without IS, show the same asymptotic growth rates.

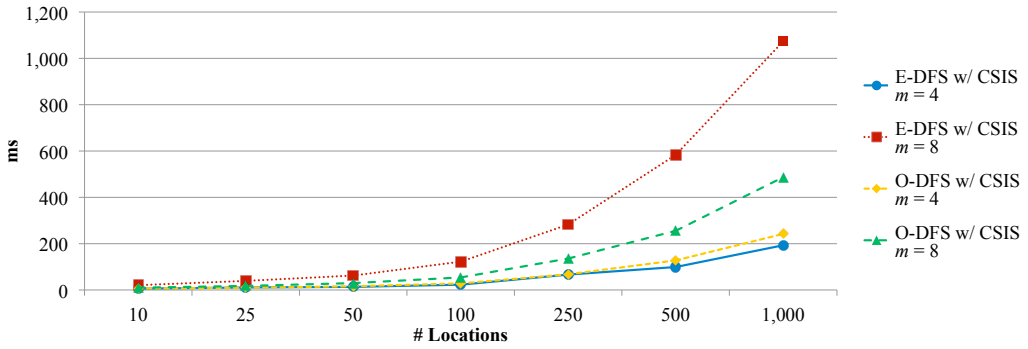


Figure 7.2: The results of Test Case 2 for the exhaustive search (E-DFS) and the ordered search (O-DFS) with $m = \{4, 8\}$.

Figure 7.2 depicts the measurements from the second test case for the exhaustive and the ordered search, with regard to their behavior facing growing search spaces. In this test case, the locations' context service infrastructures differ more strongly than in the first. Hence, the variations without the context service index structure perform even worse in comparison. As a reference, E-DFS w/o CSIS takes $40,571\ ms$ for $1,000$ locations with $m = 8$. In order to better illustrate the effects of the problem size, the figure focuses on the two variations with the index structure.

With $m = 4$, the exhaustive search slightly outperforms the ordered search, needing $100\ ms$ – compared to $128\ ms$ – for the setting with 500 locations. Hence, for smaller problems, the ordering heuristic has a negative trade-off. However, with $m = 8$, the trade-off is positive and the ordered search is the fastest with $486\ ms$ – compared to $1,074\ ms$. The overall complexity in this scenario is slightly lower than in the first one, which shows in the average runtime of all variations of $1,573\ ms$ – compared to $2,079\ ms$ – for 500 locations with $m = 4$.

Further analysis shows that the ordered search is more stable than the exhaustive search for different r values, i.e., for varying ratios of locations with and without solutions. Table 7.1 gives an overview of the respective runtime measurements. The results show that the ordering heuristic's overhead slows down the algorithm in environments

R	$r = 10\ \%$	$r = 20\ \%$	$r = 30\ \%$	Δ
E-DFS w/ CSIS	$130.5\ ms$	$255.3\ ms$	$386.5\ ms$	$256.0\ ms$
O-DFS w/ CSIS	$219.9\ ms$	$268.9\ ms$	$312.2\ ms$	$92.3\ ms$

Table 7.1: Runtime comparison of the exhaustive and the ordered search with regard to the ratio of locations with solutions $r \in R$ for $1,000$ locations.

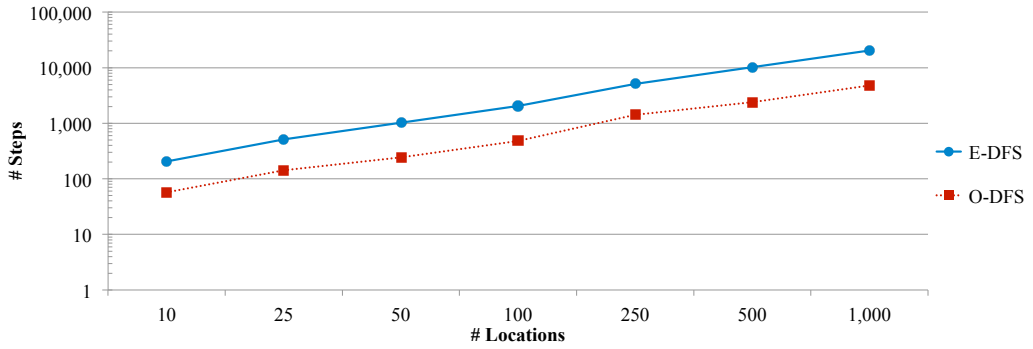


Figure 7.3: The results of Test Case 3 for the exhaustive search (E-DFS) and the ordered search (O-DFS) with $m = 4$.

with few solution containing locations, e.g., $r = 10$ %. However, with an increasing ratio of valid locations, the heuristic's benefits outweigh the overhead, resulting in a better performance of O-DFS over E-DFS starting at an r value of approximately 23 %.

The third test case has the highest complexity of the test cases with regard to the variety of capabilities in the context service infrastructure. Figure 7.3 shows the results of Test Case 3 with regard to the average number of labeling steps for the exhaustive and the ordered search. As all locations provide all necessary context services – with regard to their associated context variables, not their respective context variable states' support – and the figure depicts the number of labeling steps instead of the runtime, no differentiation between searches with and without the service index structure is applicable. In general, the ordered search outperforms the exhaustive search, consistently requiring between 23.3 and 27.8 % of E-DFS's steps.

Finally, the memory requirements of the different algorithm variations were measured using highly complex setups with 15,000 locations and 1-3 context services per location. As to be expected, O-DFS w/ CSIS has the highest memory consumption. E-DFS w/o CSIS and O-DFS w/o CSIS have the lowest memory usage, as they work without the index structure. However, even for this rather large environment, the configuration management component never exceeded 191 MB of memory usage. For smaller, more realistic search problems, the component requires approximately 35 MB of memory. Hence, the memory requirements of any of the algorithm variations is not an issue for any device that would typically be used as a central instance in a smart environment.

In summary, the similarity in the context service composition of the locations, as well as how specific/restricting the applications' requirements are, are crucial for the performance of the different algorithm variations. If the context service composition is

strongly diverse, and the application depends on a very specific composition of services, using the context service index structure is crucial. Otherwise, if the locations are equally equipped with services, and the context states required by the application are rather specific, the ordering heuristic is more beneficial than the index structure. In total, the ordered search with index structure (O-DFS w/ CSIS) performs best in almost all cases – except for a few scenarios in the second test case – at modest memory cost.

7.1.2 Adaptation Plan Algorithms

Interference resolution is by far the biggest factor in the runtime of COMITY's adaptation coordination process [65]. COMITY's original approach to interference resolution is a backtracking-based algorithm with an additional pruning mechanism that operates on the last functional combination of configurations. The approach has the benefit of finding a solution with the minimal number of adaptations. However, the approach quickly exceeds a runtime of several minutes in pervasive systems with many applications, or interferences with high complexity, respectively. Further, it terminates as soon as it has found an interference-free adaptation plan, disregarding the utilities of the individual configurations of the applications, which proactive adaptation aims to optimize. Finally, a proactive framework should pre-coordinate all viable combinations for future situations.

Hence, a new approach to interference resolution was developed for this framework with the goals of (i) finding the adaptation resolution plan that maximizes the global utility of the system, (ii) conducting a complete search for pre-coordination, and (iii) improving the runtime for a fallback reactive adaptation. In order to maximize the global utility, COMITY's context contracts were extended by a utility metric, namely the utility-cost-ratios of the adaptation alternatives. For a complete search, the algorithm simply does not terminate until it has traversed the entire search space. Finally, in order to achieve a better runtime, the new approach uses informed search techniques following [86], as well as a branch and bound-based modification of the complete search algorithm that prunes the search space. With the latter modification, the algorithm can terminate as soon as the first solution was found in case of a fallback reactive adaptation, while providing a close to optimal solution.

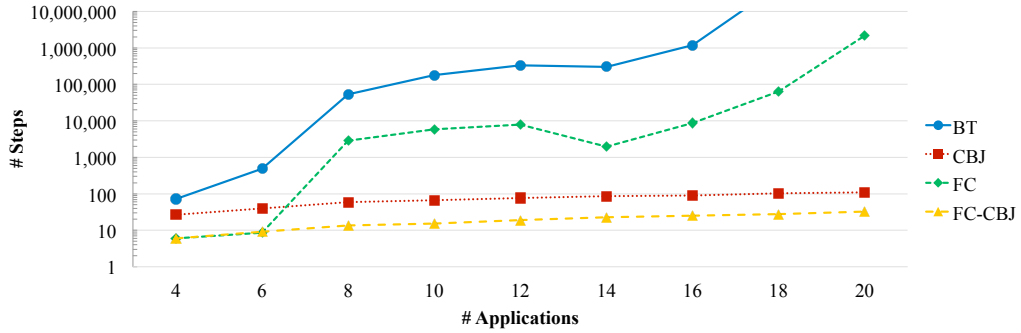
The adaptation coordination approach was evaluated on three identical desktop PCs, each with an Intel Core 2 Quad Processor Q6600 CPU (2.40 GHz per core) and 4 GB RAM, running a 64-bit Ubuntu 12.04 operating system.

In total, the adaptation coordination component features a set of eight interchangeable interference resolution procedures that implement a combination of informed search techniques and pruning, namely *backtracking* (BT), *conflict-directed backjumping* (CBJ), *explicit forward* checking (FC), *explicit forward checking with conflict-directed backjumping* (FC-CBJ), as well as their *optimal* counterparts O-BT, O-CBJ, O-FC, and O-FC-CBJ. All eight were evaluated using 50 randomly generated test cases with the following two parameters, which define the pervasive system as well as the specific interference resolution problem: (i) the number of applications $N = \{4, 6, 8, 10, 12, 14, 16, 18, 20\}$, and (ii) the number of context contracts per application that can resolve the interference $R = \{m/2, m/4\}$, with the number of context contracts per application fixed at $m = 8$. For each application, the context contracts that can resolve the interferences are distributed randomly among the entire set of context contracts. Additionally, each contract has a random utility value out of $[0.1, 1.0]$ in steps of 0.1. Further, the number of applications that minimally need to be adapted is fixed at $a = n/2$, and the number of attributes per context contract is fixed at $|CI/IS| = 5$. Initially, two of the applications are involved in an interference.

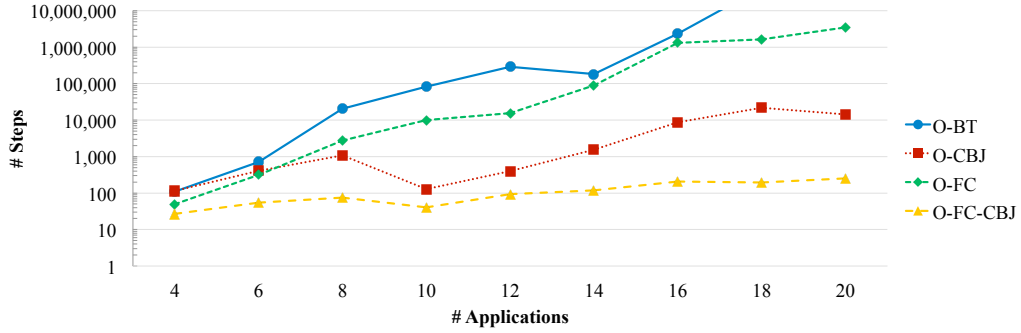
The solution space is not defined by any factors with regard to where solutions can be found. Instead, a set of applications that are not involved in the initial interference may have to be adapted as well. That is, a combination of contracts that is interference-free for the applications initially involved may result in interferences with applications previously not involved. As a result, the search space becomes unpredictable.

Figure 7.4 shows the average number of steps, i.e., the number of consistency checks that are executed, required by each variation of the two sets of resolution algorithms with respect to the number of applications n and the size of the solution space defined by $r = m/2$. The number of steps is used as the first unit of measurement to show the relation between the different algorithms, regardless of the specific system they are running on. The runtime in ms is discussed subsequently. As expected, FC-CBJ and O-FC-CBJ, respectively, outperform the other algorithms every time. For 20 applications with 8 context contracts each – i.e., a total of 8^{20} possible combinations – FC-CBJ performs, on average, 32.66 consistency checks with $r = m/2$ and 33.86 with $r = m/4$, respectively, in order to find an interference-free combination. While searching for the combination with the maximum global utility, O-FC-CBJ performs, on average, 250.3 and 265.7 consistency checks, respectively.

Figure 7.4a shows the traditional set of informed search algorithms that terminate as soon as the first solution was found for high responsiveness. All algorithms show a runtime exponential in n, with some noise due to randomness. However, the growth rate



(a) Informed Search Algorithms for High Responsiveness

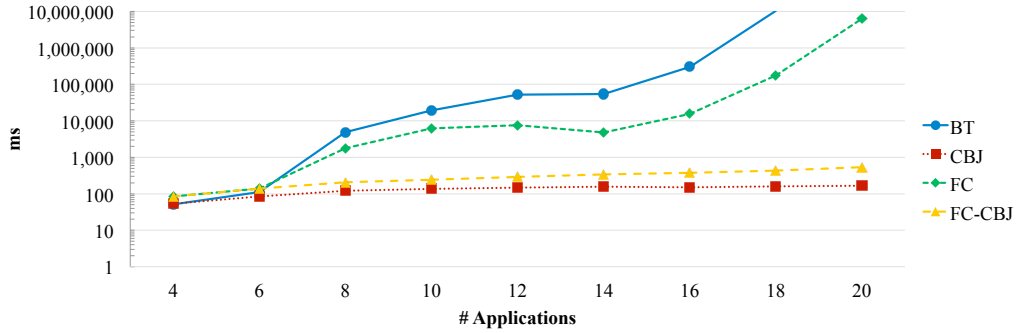


(b) Branch and Bound Algorithms for Maximizing Utility

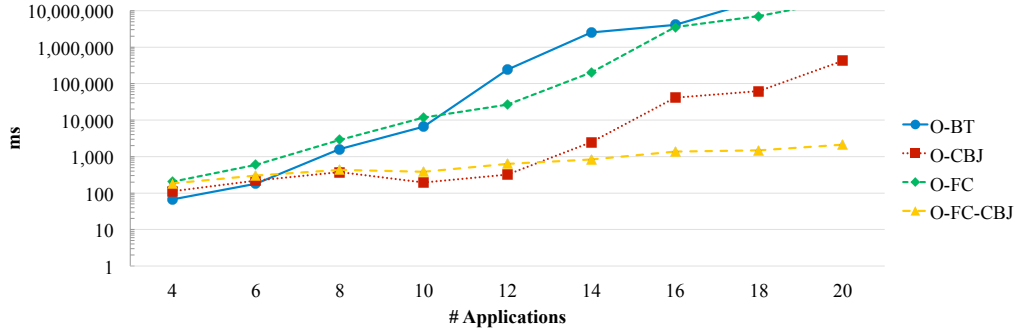
Figure 7.4: The results of the performance measurements for the two sets of interference resolution algorithms in # *Steps* with regard to the number of applications sharing their context and with $r = m/2$.

of CBJ and FC-CBJ is substantially lower than that of BT and FC, respectively. Additionally, the graphs suggest strong ties between the performances of those algorithms using the same backtracking strategy. That is, BT and FC, which both backtrack only one level in case of an inconsistent subsolution, share growth characteristics, as do CBJ and FC-CBJ, which both do a conflict-directed backjump in case of an inconsistency. Hence, it is safe to conclude that the level of *informed backtracking* has the most significant effect on the performance of the algorithms. On average, the algorithms combined find interference resolution plans with between 1.16 and 1.2 additional adaptations compared to the number that is minimally required.

Figure 7.4b shows the set of algorithms for maximizing the global utility. The performance characteristics of these four algorithms is similar to those terminating after they find the first solution. One interesting finding is that with O-CBJ, the effect of the pruning heuristic greatly improves with $n \geq 10$. With less than 10 applications,



(a) Informed Search Algorithms for High Responsiveness



(b) Branch and Bound Algorithms for Maximizing Utility

Figure 7.5: The results of the performance measurements for the two sets of interference resolution algorithms in ms with regard to the number of applications sharing their context and with $r = m/2$.

the heuristic – even though the smallest possible overestimate – does not prune much of the relative search space, which shows in the number of consistency checks. Further, comparing O-CBJ with O-FC-CBJ, the explicit forward checking strategy proves to have a very positive effect while searching for the maximum global utility.

Figure 7.5 shows the average runtime in ms required by each variation of the two sets of resolution algorithms with respect to the number of applications n and the size of the solution space defined by $r = m/2$. Using this metric, the growth characteristics are, as one would expect, the same as with the average number of steps. However, CBJ actually outperforms the other algorithms when it comes to finding a solution as fast as possible. This is due to the significant forward checking overhead of FC-CBJ. For 20 applications, CBJ takes 0.17 ms with $r = m/2$ and 0.21 ms with $r = m/4$, respectively, in order to find an interference-free combination. When searching for the solution with the maximum global utility, O-CBJ initially also proves to be the

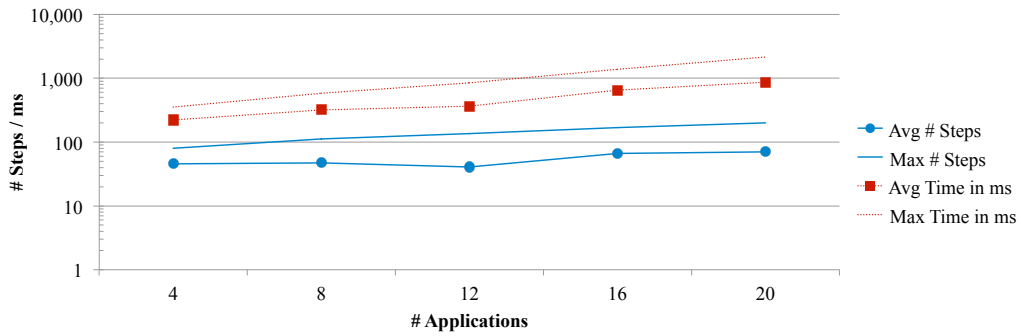


Figure 7.6: The performance of FC-CBJ in situations without solutions in the search space with regard to the number of applications sharing their context.

fastest approach. However, O-FC-CBJ eventually outperforms O-CBJ with $n \geq 14$ and shows a significantly smaller growth rate. Hence, after this point, the positive effect of the forward checking strategy outweighs its overhead. For 20 applications, O-FC-CBJ takes, on average, 2.1 s to find the optimal solution. Further, it finds the optimal solution in less than one second for as many as 14 applications.

Finally, Figure 7.6 shows the average and maximum runtime (in # *Steps* and ms, respectively) of the best performing FC-CBJ in situations where there are no solutions with $n = \{4, 8, 12, 16, 20\}$. For up to 20 applications in the environment, the interference resolution process determines such an *unresolvable* situation consistently in less than one second, with a maximum value of $2,157$ ms. In these situations, a manual adaptation or the termination of an application, respectively, becomes necessary.

Overall, the results show a significant improvement in runtime compared to COMITY's original approach (cf. [65]) – including the algorithms maximizing the global utility – at the cost of not finding the adaptation plan with the minimal number of adaptations. However, this is not necessarily a drawback, as proactive adaptation tries to pre-coordinate adaptations, i.e., at a point in time where there is no actual starting configuration to work with.

Next, the entire framework for proactive adaptation is evaluated in extensive simulations using recorded user traces with regard to the number of successful pre-calculations, the runtimes of the individual adaptation control loops, as well as system utilization metrics.

7.2 Simulation

In this section, the prototype implementation of the proactive adaptation framework is applied in real-time simulations. First, the evaluation setup is described. Afterwards, Section 7.2.2 presents various measurement results, including times for proactive and fallback reactive adaptations, before Section 7.2.3 examines different system load characteristics, such as memory consumption and CPU load.

7.2.1 Simulation Environment

In order to extensively evaluate the framework for proactive adaptation, as well as the concept of proactive adaptation in general, an interactive environment with user/application movement is simulated. The environment itself consists of 20 locations with up to ten context variables each. In it, all context variables have the same parameterized number of possible states.

Up to ten applications move through the environment based on a modified set of real user traces from the *Augsburg Indoor Location Tracking Benchmarks* project [81, 82]. The original set of traces contains recordings for four users. It was enhanced to ten by duplicating the traces of the three most active users twice – the record of the forth user is considerably smaller than that of the other three – while substituting their most frequent location, presumably their respective offices, with new locations. Further, the traces of the six new users were modified by introducing small random time shifts up to a few minutes. As a result, the data maintains its patterns of daily workplace interaction, such as meetings or joint coffee breaks. That is, the users frequently share the same location, but arrive and leave shorty after each other, creating the situations in which the users' applications need to adapt.

Figure 7.7 shows the floor plan of the simulated pervasive environment following the floor plan in [82] – with adjustments for the additional locations. Further, the figure visualizes the number of visits per user and location. Naturally, the recorded number of location events per user for the corridor is inflated, as the users have to pass through the corridor in order to get from one location to another. Aside from the corridor, the data shows common gathering points that are ideal for coordinating adaptation plans, e.g., room 402 and the kitchen, as well as locations that are rarely visited by more than one user, e.g., rooms 403 and 409. Of the user traces data set, half was used for training the predictors, and the other half fed into the system at runtime.

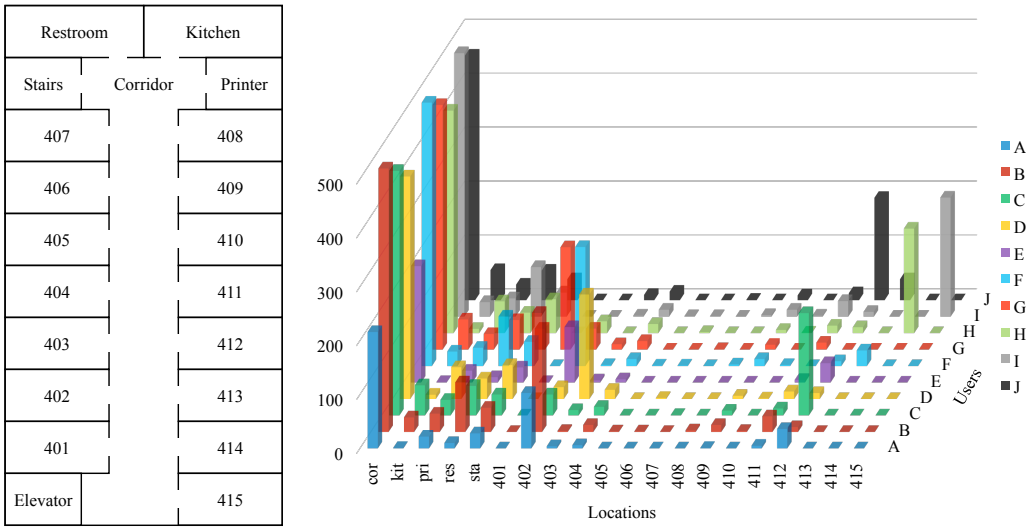


Figure 7.7: The floor plan of the simulated environment (left) and number of visits per user and location (right).

The applications' requirements are randomly generated before each run based on the following parameters: (i) the minimal and maximal number of terms per requirement t_{min} and t_{max}, (ii) the minimal and maximal number of literals per term l_{min} and l_{max}, and (iii) the minimal and maximal number of states per literal s_{min} and s_{max}. The theoretical effects of these three parameters on the simulation are somewhat conflicting. While larger *min* and *max* values, especially for t and l, create bigger search spaces for the adaptation alternative and adaptation plan algorithms, they also bring a higher potential for requirement overlappings, and thus more possible solutions. In contrast, smaller *min* and *max* values, this time especially for s, keep the search space small, but also reduces the potential for interference-free sets of adaptation alternatives.

All simulations were conducted on a standard desktop PC with an Intel Core i5-2500K CPU (four cores with 3.3 GHz each), 8 GB of main memory, and a 64-bit Windows 8 operating system. Each entity in the simulations – i.e., each application, sensor, actuator, and predictor, as well as the context broker and the adaptation control component – ran in its own Java Virtual Machine on top of BASE, thereby emulating a completely distributed system, albeit without network delays. From the two sets of algorithms, E-DFS w/o CSIS and O-FC-CBJ were used. The choice of E-DFS w/o CSIS for the simulation has several reasons. First, the ordering heuristic of O-DFS would have no benefit, as all context variables have the same amount of possible states, resulting in equal complexity indexes. Second, the index structure – as beneficial as it is by greatly reducing the necessary interaction between the components in the system – is

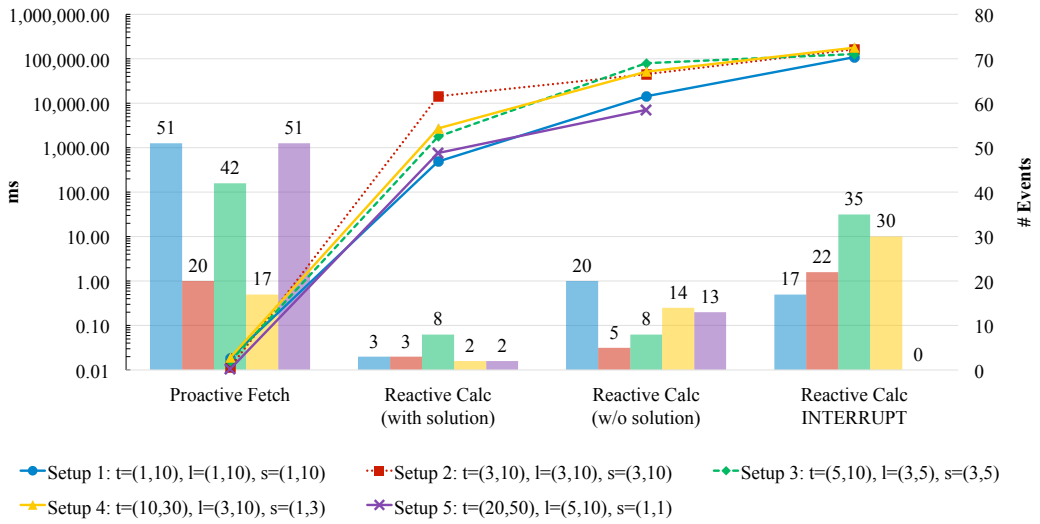
not crucial for smaller environments, while its omission provides an increase in system load that is desirable in evaluation settings. Finally, in order to be able to accelerate the simulations that are restricted by the nature of real time data – thus creating long periods of idling – a timer was used that runs x-times as fast as the host's internal clock. In the simulation runs that produced the forthcoming results, a factor of ten was used for all runs, i.e., a ten hour period was simulated in one hour.

7.2.2 Simulation Results

Evaluating real time systems differs from conducting algorithm performance measurements significantly. The simulations are time consuming and time restricted at the same time, regardless of the complexity of individual problems, and the individual executions of procedures do not happen in isolation. That is, on the one hand, the calculations triggered by a certain event – which is not under the developer's control but originates from inside the system itself – may terminate very fast, resulting in idle time. On the other hand, in case the calculations have not yet come to a conclusion before the next event happens, time as a resource becomes the issue. In such situations, the new task can either be queued for later execution, started in a new thread – thereby creating an additional competitor for the system's resources – or replace the ongoing task. As described in Section 5.6.6, the adaptation control component uses all three alternatives in different situations. Proactive adaptation alternative searches are queued for execution in application-related threads, whereas proactive adaptation coordination tasks are queued for execution in location-related threads. All reactive calculations, in contrast, replace ongoing calculations, as the result is needed as fast as possible, and the result of the old task automatically becomes obsolete for that reactive adaptation control cycle. During the conducted simulations, such task replacements were recorded as *reactive calculation interrupts*.

In total, 45 simulations were conducted with varying parameter combinations, random application requirements generated from these combinations, and differing user movement patterns from distinct days in the data set. Figure 7.8 depicts the results of a sample of simulation runs with medium complexity. For comparability, all runs simulate the same workday, i.e., user movement is the same in each run. Further, each location has the same ten context variables, with ten possible states each. However, the runs differ in their setups regarding the parameters t, l, and s as follows:

- Setup 1: $t = [1, 10]$, $l = [1, 10]$, $s = [1, 10]$
- Setup 2: $t = [3, 10]$, $l = [3, 10]$, $s = [3, 10]$



Setup	Proactive Fetch	Reactive Calculation		
		with solution	w/o solution	INTERRUPT
1	0.02 ms	490.33 ms	14,366.55 ms	109,380.88 ms
2	0.01 ms	14,302.67 ms	45,411.60 ms	163,981.32 ms
3	0.02 ms	1,764.13 ms	79,997.25 ms	128,477.17 ms
4	0.02 ms	2,707.00 ms	51,512.64 ms	177,285.60 ms
5	0.01 ms	759.00 ms	7,137.46 ms	n/a

Figure 7.8: The evaluation results with regard to the average runtime in ms and the respective number of events for five different simulation setups.

- Setup 3: $t = [5, 10]$, $l = [3, 5]$, $s = [3, 5]$
- Setup 4: $t = [10, 30]$, $l = [3, 10]$, $s = [1, 3]$
- Setup 5: $t = [20, 50]$, $l = [5, 10]$, $s = [1, 1]$

The results clearly indicate the benefits of the proactive scheme. In all five simulations, fetching a pre-calculated adaptation plan from the plan base takes, on average, between 0.01 and 0.02 ms, with an absolute minimum of 0.0056 ms, and an absolute maximum of 0.0625 ms. In contrast, the average time for reactively calculating an adaptation plan varies from 0.5 to 14.3 s, with an absolute minimum of 45.5 ms, and an absolute maximum of 30.2 s, constituting a human-perceivable delay that is undesirable in pervasive environments. Moreover, in case there are no interference-free adaptation plans in the search space, the reactive calculations terminate after, on average, 39.7 s, with an absolute maximum of 530.6 s. Finally, the interrupted reactive calculations were replaced after, on average, 144.8 s. At this point the timer is reset to measure the runtime of the new cycle (proactive or reactive), as the situation has changed.

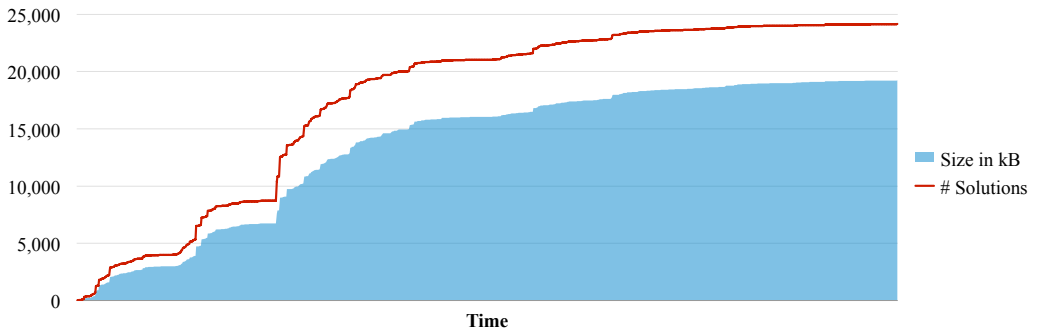


Figure 7.9: Growth of the plan base during simulation of Setup 1 in terms of the number of solutions and kB of memory.

The main factors in the calculation times are the number of users at the same location, and the level of diversity of the applications' requirements with regard to the shared context. As a result, the time difference between a proactive fetch and a reactive calculation increases significantly in runs with higher complexity. However, the share of situations without solutions increases as well. In order to be able to compare the proactive and reactive cycle times, prior knowledge about such unsolvable situations originating from pre-coordination attempts was discarded during simulation. In a productive system, this knowledge can be used to avoid a share of the unsuccessful reactive cycles.

Regardless, during a reactive cycle, there is no indication whether it will be successful or not. In case the system is not able to fetch a pre-calculated adaptation plan, the users are forced to wait until the reactive calculation terminates with or without a solution, including possibly several interruptions. The reactive calculations – with or without a solution – terminate after an average of 95.1 s with, and 26.1 s without including interrupts.

Every solution calculated at runtime, regardless of proactive or reactive cycle, is stored in the plan base. Figure 7.9 shows the growth of the plan base during a simulation run of Setup 1 with regard to the number of coordinated adaptation plans it contains, as well as the size of the data structure in memory. After simulating the ten hour business day period, the plan base contained $24,161$ coordinated adaptation plans, occupying $19,233$ kB of memory, and seemingly converging to a maximum. For evaluation purposes, the plan base was reseted for each run. In a productive system, the adaptation plans should remain in the plan base for some time, so that they do not need to be re-calculated too often.

Finally, assuming the interrupted calculations would not have terminated with solutions, the prototype pre-calculated adaptation plans for 91 % of the situations, in which a coordinated adaptation was necessary and possible. This share was achieved using the very generic context prediction algorithms implemented in the prototype. It stands to reason that the share can be further increased through tailoring the predictors and the prediction process, e.g., using aggregation, validation, etc., to the specific environment.

7.2.3 System Utilization

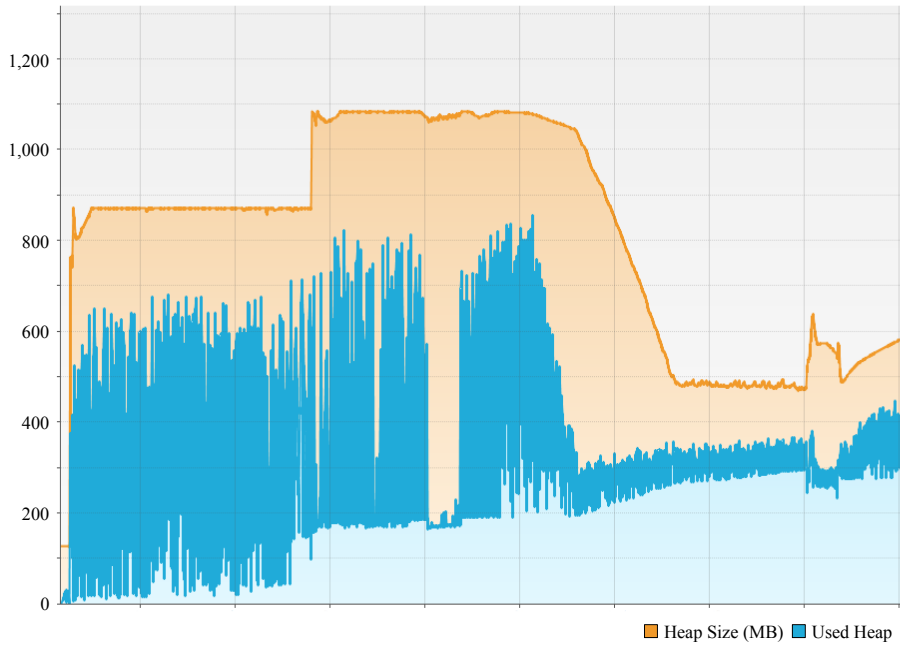
Figure 7.10 shows the results of monitoring the Java VM using VisualVM 1.3.8 [70] during a simulation run with the same parameters as Setup 1 (cf. Section 7.2.2). More specifically, Figure 7.10a depicts the heap space utilization of the configuration management and adaptation coordination component, which run in the same Java Virtual Machine. The blue line, which corresponds to the heap space that is actually used, fluctuates heavily and seemingly forms areas in the graph. This shows the amount and size of the search spaces that both the adaptation alternative and the adaptation plan algorithms create, traverse and destroy each time. At its biggest size, the heap requires just below 1.1 GB of memory. Figure 7.10b depicts the CPU usage and the activity of Java's Garbage Collector. The CPU-related graph fluctuates heavily as well, correlated to the number of simultaneous adaptation alternative and adaptation plan calculations. During simulation, the CPU usage rate spikes at approximately 93 %.

Further, both graphs show parallel phases of high activity, namely a jump in heap size, and a significantly high workload in case of the CPU usage, respectively. Moreover, Figure 7.9 shows a rapid increase in the number of adaptation plans in the plan base. These extremes coincide with a phase of high user mobility between 10 a.m. and 1 p.m. The phase of high mobility in the afternoon, however, is not reflected in the graphs, suggesting a high rate of pre-computed adaptation plans.

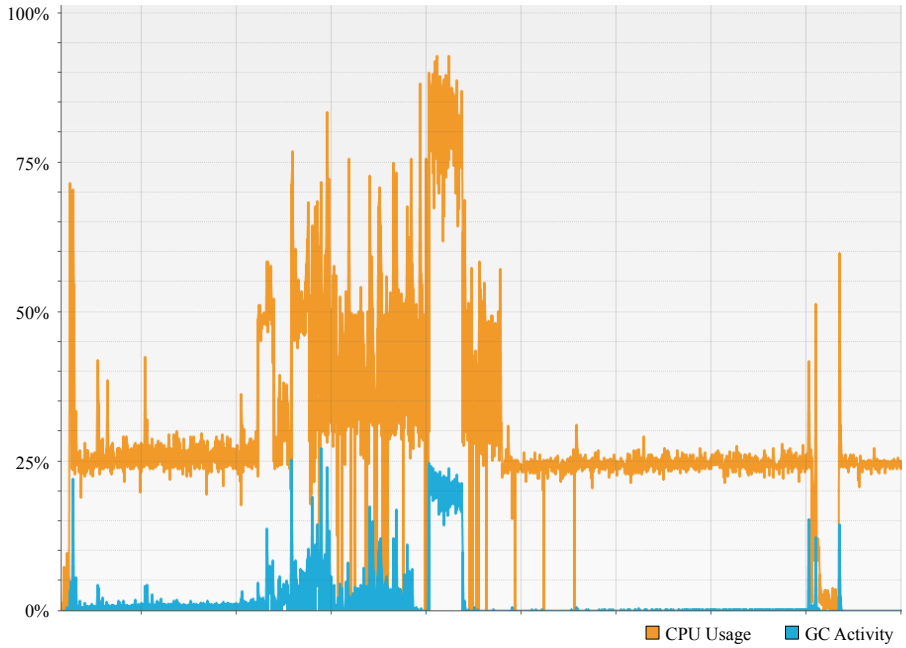
Overall, the measurements show the necessity of centralized components in the system in order to run the complex configuration and coordination algorithms.

7.3 Discussion

Adapting an application or system based on pre-calculated adaptation plans is, in theory, obviously faster than having to calculate the plan first. Through prior



(a) Heap Utilization



(b) CPU Usage and Garbage Collector Activity

Figure 7.10: Output of the Java VM monitoring using VisualVM during a simulation with the parameters $t = [1, 10]$, $l = [1, 10]$, and $s = [1, 10]$.

calculation, adaptations become scripted, resulting in adaptation times comparable to those in static environments. The simulation results proved that this assumption holds true for the prototype implementation of the framework developed in this thesis. However, the goal of the evaluation, and the thesis in total, was rather to show how the benefits of the proactive scheme can be achieved in dynamic environments, and at what cost.

The algorithm performance analysis in Section 7.1 showed that the CSP-based approach to modeling and solving the two problems of finding all possible adaptation alternatives of an application, as well as coordinating the adaptation alternatives of multiple applications in the same context, while optimizing the system's global utility, is a viable solution for real-time systems. The algorithms scale well enough with regard to the problem size for medium to large sized pervasive systems, i.e., several hundred locations with up to 20 applications in the same context, with average runtimes of at most 2.1 *s* for the fastest algorithm variations in the largest settings.

The simulations conducted in Section 7.2 then showed that the prototype framework is applicable for small to medium sized, office-like environments with ten users and 20 locations. For over 90 % of the total adaptation situations, the framework was able to fetch a pre-calculated adaptation plan instantly, avoiding reactive adaptation calculation times of 20 seconds and more. In the process, the total workload was high at times with almost 1.1 GB of reserved heap space and over 90 % CPU usage rate, but manageable for a standard desktop PC. However, larger environments should be managed by more than one central entity.

Beyond these measurements, the simulations provide other observations to reason about. First, the runtime measurements from the isolated algorithm performance analysis are not applicable in live systems. While they indicate a certain level of performance and are valuable to examine the algorithm's asymptotic growth, they also create false expectations. In a live system, these algorithms are often executed in parallel to others, and are typically part of a larger process with overhead from communication and concurrency issues. Second, the proactive scheme is not only beneficial in situations with possible solutions. Prior knowledge about unsolvable situations, i.e., in case a proactive calculation terminates with no solutions, can be used to avoid these in the short run, and, even more so, be applied to analyzing the system in order to identify the lack of certain services in the infrastructure or incompatible applications, making the environment more intelligent. Avoiding unsolvable situations ties in to the final observation. The simulation results showcase the importance of accurate context prediction to the concept, and, at the same time, how to compensate for bad predictions.

The prototype framework achieved a high rate of pre-calculated adaptation plans using generic pattern matching and Markov Model-based predictors. However, the rate was increased by using the top three predictions in terms of similarity score and transition probability, respectively, leading to unnecessary calculations and additional workload. Further, the framework stores all calculated adaptation plans for later retrieval, instead of discarding them after the adaptation, which can cover up false predictions. Now, avoiding unsolvable situations, e.g., by rerouting a user, can decrease the accuracy of predictions, as the instructed behavior may not be consistent with the natural behavior of the entity. Nevertheless, prediction accuracy can also be improved by tailoring the approaches to the specific environment/prediction subject, as well as aggregating the results of multiple approaches. Case in point, a pattern matching algorithm can miss on a certain transition, whereas a probability-based approach struggles with predicting rare events.

In summary, the evaluation of the prototype framework showed that the concept of proactive adaptation in dynamic pervasive environments is not only beneficial in theory, but feasible in practice given the necessary system support. The next chapter closes this thesis with a conclusion and an outlook on future work.

8 Conclusion and Outlook

The previous chapter evaluated the prototype framework for proactive adaptation regarding the performance of the search algorithms and the framework's behavior in real-time, multi-user pervasive systems. This chapter closes the present thesis with a conclusion and an outlook on future work.

8.1 Conclusion

Next to the traditional challenges of dynamic distributed systems, such as heterogeneous and changing resources, proactive adaptation in multi-user pervasive systems requires a constant effort of monitoring context information and predictions, as well as calculating, coordinating, and instructing adaptations. Without suitable architectures and algorithms, these challenges are hard to overcome. In order to enable proactive adaptation, this thesis presented a comprehensive framework including middleware-based system support for automatic coordinated application-level proactive adaptation in multi-user pervasive systems.

The framework's context interaction model and corresponding context broker mask the heterogeneity of resources by offering uniform access to context sensing, predicting, and actuating services based on the abstraction of context variables. Additionally, the broker's dynamic service registry handles the changing environment transparently for applications. Further, the framework reduces the complexity of application development by assuming the responsibility of calculating and rating possible adaptation alternatives of an application with regard to its predicted context. For this, an application configuration model was developed that allows applications to specify their requirements towards their context, including user preferences. Moreover, the model introduced duration-dependent utility and cost functions, in order to optimize adaptation decisions with regard to the future context time series.

The thesis presented two sets of algorithms in order to predetermine interference-free adaptation plans. First, for adaptation pre-calculation, the problem of finding adaptation alternatives was constructed as a constraint satisfaction problem, and an

exhaustive backtracking-based algorithm including ordering heuristic was developed that finds all valid application configurations. Second, for adaptation pre-coordination, the thesis presented a set of tree-based interference resolution algorithms for the application coordination framework COMITY that are able to optimize the global utility of the environment. Equipped with this new set of algorithms, COMITY was then integrated into the framework by mapping application requirements to context contracts, as well as constructing virtual environments for its interference detection approach.

To complete the framework, its individual models and components were integrated in an adaptation control loop for proactive adaptation. The control component monitors current and predicted context, triggers calculations and coordinations, manages adaptation plans, and instructs adaptations in a proactive, as well as a fallback reactive loop. With the presented framework, context-aware applications can participate in proactive adaptive systems by specifying their configuration model and following adaptation instructions.

Finally, the evaluation showed that – given suitable system support – proactive adaptation in multi-user pervasive systems is both feasible and beneficial at the modest cost of a centralized entity in the class of a standard desktop PC. Most importantly, the framework helps to reduce adaptation delays and avoids the oscillating effects caused by context interferences through pre-calculation and pre-coordination, respectively.

8.2 Outlook

There are several interesting research challenges that have emerged during the development of this thesis and are worth further exploration.

First, the application configuration model could be extended to feature service composition conditional utility and cost functions. That is, instead of accumulating the application configuration ratings from the individual functions per context variable, the functions could depend on specific ensembles of context services. As a result, the ratings would become service composition-dependent themselves, which may reflect the actual utilities and costs of a configuration more accurately.

Second, in order to make the adaptation search process more efficient, the adaptation control component could utilize the fact that a specific CSP for finding adaptation alternatives or adaptation plans is unsolvable, which is known after an unsuccessful search. Additionally, this extension could be used to identify incompatible application

constellations, which, with this knowledge, could be addressed by application developers and users.

Third, with regard to optimizing adaptations, there is a non-trivial conflict between the best series of adaptations for a single application and the maximal global utility of the system. Adaptation series for a single application can, for example, be optimized using adaptation strategies and decision trees. Optimization of adaptation series for multiple applications, on the other hand, requires additional considerations, such as group mobility and compatibility of individual adaptation strategies.

Finally, enhancements to the theoretical framework could potentially minimize the number of unsolvable situations and should be explored. Possible additions could be metrics for measuring/estimating the *distance* between application configurations, adaptation plans, and context situations, integration of negotiation protocols, as well as the concept of compromise.

Bibliography

[1] G. D. Abowd, A. K. Dey, R. Orr, and J. Brotherton. Context-awareness in Wearable and Ubiquitous Computing. *Virtual Reality*, volume 3, number 3, pages 200–211, 1998.

[2] S. Ahn and D. Kim. Proactive Context-aware Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN)*, pages 38–53, Springer, 2006.

[3] M. Alia, M. Beauvois, Y. Davin, R. Rouvoy, and F. Eliassen. Components and Aspects Composition Planning for Ubiquitous Adaptive Services. In *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 231–234, IEEE, 2010.

[4] Autonomic Computing Group. An Architectural Blueprint for Autonomic Computing. *IBM White paper*, 2005.

[5] M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context-aware Systems. *International Journal of Ad Hoc and Ubiquitous Computing*, volume 2, number 4, pages 263–277, 2007.

[6] M. Bauer, C. Becker, and K. Rothermel. Location Models from the Perspective of Context-aware Applications and Mobile Ad Hoc Networks. *Personal and Ubiquitous Computing*, volume 6, number 5-6, pages 322–328, 2002.

[7] C. Becker. *System Support for Context-aware Computing.* Habilitation, Fakultät 5: Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, 2004.

[8] C. Becker and F. Dürr. On Location Models for Ubiquitous Computing. *Personal and Ubiquitous Computing*, volume 9, number 1, pages 20–31, 2005.

[9] C. Becker, M. Handte, G. Schiele, and K. Rothermel. PCOM – A Component System for Pervasive Computing. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (PerCom)*, pages 67–76, IEEE, 2004.

[10] C. Becker and D. Nicklas. Where Do Spatial Context-Models End and Where Do Ontologies Start? A Proposal of a Combined Approach. In *Proceedings of the First International Workshop on Advanced Context Modeling, Reasoning and Management at the Sixth International Conference on Ubiquitous Computing (UbiComp)*, 2004.

[11] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. BASE – A Micro-Broker-based Middleware for Pervasive Computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 443–451, IEEE, 2003.

[12] R. Begleiter, R. El-Yaniv, and G. Yona. On Prediction Using Variable Order Markov Models. *Journal Artificial Intelligence Research*, volume 22, pages 385–421, 2004.

[13] E. Bergman. *Information Appliances and Beyond: Interaction Design for Consumer Products*, Morgan Kaufmann, 2000.

[14] C. Bertolli, G. Mencagli, and M. Vanneschi. A Cost Model for Autonomic Reconfigurations in High-Performance Pervasive Applications. In *Proceedings of the Forth ACM International Workshop on Context-Awareness for Self-Managing Systems (CASEMANS)*, ACM, 2010.

[15] A. Boytsov and A. Zaslavsky. Extending Context Spaces Theory by Proactive Adaptation. *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 1–12, 2010.

[16] A. Boytsov and A. Zaslavsky. Context Prediction in Pervasive Computing Systems: Achievements and Challenges. In *Supporting Real Time Decision-Making*, pages 35–63, Springer, 2011.

[17] P. J. Brown. Triggering Information by Context. *Personal Technologies*, volume 2, number 1, pages 18–27, 1998.

[18] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications*, volume 4, number 5, pages 58–64, 1997.

[19] C. J. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, volume 2, number 2, pages 121–167, 1998.

[20] H. E. Byun and K. Cheverst. Supporting Proactive 'Intelligent' Behaviour: The Problem of Uncertainty. In *Proceedings of the User Modeling (UM) Workshop on User Modeling for Ubiquitous Computing*, pages 17–25, 2003.

[21] A. T. S. Chan and S.-N. Chuang. MobiPADS: A Reflective Middleware for Context-aware Mobile Computing. *IEEE Transactions on Software Engineering*, volume 29, number 12, pages 1072–1085, 2003.

[22] C. Chang, S. Ling, and S. Krishnaswamy. ProMWS: Proactive Mobile Web Service Provision Using Context-awareness. In *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 69–74, IEEE, 2011.

[23] G. Chen, D. Kotz, and Others. A Survey of Context-aware Mobile Computing Research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, 2000.

[24] H. Chen, T. Finin, and A. Joshi. The SOUPA Ontology for Pervasive Computing. In *Ontologies for Agents: Theory and Experiences*, pages 233–258, Springer, 2005.

[25] H. L. Chen. *An Intelligent Broker Architecture for Pervasive Context-aware Systems*. PhD thesis, Department of Computer Science and Electrical Engineering, University of Maryland, 2004.

[26] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 17–24, ACM, 2000.

[27] D. J. Cook, M. Youngblood, E. O. Heierman III, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. MavHome: An Agent-based Smart Home. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 521–524, IEEE, 2003.

[28] J. R. Cooperstock, K. Tanikoshi, G. Beirne, T. Narine, and W. A. Buxton. Evolution of a Reactive Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 170–177, ACM, 1995.

[29] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of the IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pages 806–810, IEEE, 2005.

[30] N. Davies, K. Mitchell, K. Cheverst, and G. Blair. Developing a Context Sensitive Tourist Guide. In *Proceedings of the First Workshop on Human Computer Interaction with Mobile Devices (MobileHCI)*, 1998.

[31] A. K. Dey. Context-aware Computing: The CyberDesk Project. In *Proceedings of the 1998 AAAI Spring Symposium on Intelligent Environments*, pages 51–54, AAAI, 1998.

[32] A. K. Dey and G. D. Abowd. Towards a Better Understanding of Context and Context-awareness. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing (HUC)*, Springer, 1999.

[33] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems*, volume 1, number 2, pages 223–259, 2006.

[34] W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 279–286, ACM, 2002.

[35] G. Fenza, D. Furno, and V. Loia. Enhanced Healthcare Environment by Means of Proactive Context-aware Service Discovery. In *Proceedings of the 2011 IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 625–632, IEEE, 2011.

[36] A. Ferscha, M. Hechinger, R. Mayrhofer, and R. Oberhauser. A Light-weight Component Model for Peer-to-Peer Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCS Workshops)*, pages 520–527, IEEE, 2004.

[37] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned Remote Execution for Pervasive Computing. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 61–66, IEEE, 2001.

[38] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, volume 23, number 2, pages 62–70, 2006.

[39] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Toward Distraction-free Pervasive Computing. *IEEE Pervasive Computing*, volume 1, number 2, pages 22–31, 2002.

[40] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A Comprehensive Solution for Application-level Adaptation. *Software: Practice and Experience*, volume 39, number 4, pages 385–422, 2009.

[41] A. S. Goldberger. Best Linear Unbiased Prediction in the Generalized Linear Regression Model. *Journal of the American Statistical Association*, volume 57, number 298, pages 369–375, 1962.

[42] R. Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, volume 3, number 3, pages 22–30, 2004.

[43] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall. Programming for Pervasive Computing Environments. Technical Report UW-CSE-01-06-01, Department of Computer Science and Engineering, University of Washington, 2001.

[44] T. Gu, H. K. Pung, and D. Q. Zhang. A Service-oriented Middleware for Building Context-aware Services. *Journal of Network and Computer Applications*, volume 28, number 1, pages 1–18, 2005.

[45] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang. An Ontology-based Context Model in Intelligent Environments. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS)*, pages 270–275, 2004.

[46] M. Handte, G. Schiele, V. Matjuntke, C. Becker, and P. J. Marrón. 3PC: System Support for Adaptive Peer-to-Peer Pervasive Computing. *ACM Transactions on Autonomous and Adaptive Systems*, volume 7, number 1, pages 10, 2012.

[47] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay. Adaptable Pervasive Flows – An Emerging Technology for Pervasive Adaptation. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 108–113, IEEE, 2008.

[48] C. K. Hess, M. Román, and R. H. Campbell. Building Applications for Ubiquitous Computing Environments. In *Pervasive Computing*, pages 16–29, Springer, 2002.

[49] B. L. Hickman and C. L. Corritore. Computer Literacy: The Next Generation. *Computer Science Education*, volume 6, number 1, pages 49–66, 1995.

[50] D. Hilbert and W. Ackermann. *Principles of Mathematical Logic*, volume 69, American Mathematical Society, 1950.

[51] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schwehm. Next Century Challenges: Nexus – An Open Global Infrastructure for Spatial-aware Applications. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 249–255, ACM, 1999.

[52] J. Hong, E.-H. Suh, J. Kim, and S. Kim. Context-aware System for Proactive Personalized Service Based on Context History. *Expert Systems with Applications*, volume 36, number 4, pages 7448–7457, 2009.

[53] J.-Y. Hong, E.-H. Suh, and S.-J. Kim. Context-aware Systems: A Literature Review and Classification. *Expert Systems with Applications*, volume 36, number 4, pages 8509–8522, 2009.

[54] E. Horvitz, P. Koch, C. M. Kadie, and A. Jacobs. Coordinate: Probabilistic Forecasting of Presence and Availability. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 224–233, Morgan Kaufmann Publishers Inc., 2002.

[55] W. H. Hsu, N. D. Gettings, V. E. Lease, Y. Pan, and D. C. Wilkins. Heterogeneous Time Series Learning for Crisis Monitoring. In *Proceedings of the AAAI Workshop on Predicting the Future: AI Approaches to Time-Series Problems*, volume 98, pages 34–41, 1998.

[56] B. Johanson, A. Fox, and T. Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, volume 1, number 2, pages 67–74, 2002.

[57] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, volume 36, number 1, pages 41–50, 2003.

[58] K. E. Kjær. A Survey of Context-aware Middleware. In *Proceedings of the 25th IASTED International Multi-Conference on Software Engineering (SE)*, pages 148–155, ACTA, 2007.

[59] T. Kohonen. The Self-organizing Map. *Proceedings of the IEEE*, volume 78, number 9, pages 1464–1480, 1990.

[60] J. Krumm. A Markov Model for Driver Turn Prediction. Technical Report 2008-01-0195, SAE International, 2008.

[61] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A Survey on Engineering Approaches for Self-adaptive Systems. *Pervasive and Mobile Computing*, volume 17, 2014.

[62] A. Leonhardi. *Architektur eines verteilten skalierbaren Lokationsdienstes.* PhD thesis, Fakultät 5: Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, 2003.

[63] V. Majuntke. *Application Coordination in Pervasive Systems.* PhD thesis, Fakultät für Betriebswirtschaftslehre, Universität Mannheim, 2012.

[64] V. Majuntke, G. Schiele, K. Spohrer, M. Handte, and C. Becker. A Coordination Framework for Pervasive Applications in Multi-User Environments. In *Proceedings of the 2010 Sixth International Conference on Intelligent Environments (IE)*, pages 178–184, IEEE, 2010.

[65] V. Majuntke, S. VanSyckel, D. Schafer, C. Krupitzer, G. Schiele, and C. Becker. COMITY: Coordinated Application Adaptation in Multi-Platform Pervasive Systems. In *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 11–19, IEEE, 2013.

[66] R. Mayrhofer. *An Architecture for Context Prediction.* PhD thesis, Technisch-Naturwissenschaftliche Fakultät, Johannes Kepler Universität Linz, 2004.

[67] H. Mukhtar, D. Belaïd, and G. Bernard. A Quantitative Model for User Preferences Based on Qualitative Specifications. In *Proceedings of the 2009 International Conference on Pervasive Services (ICPS)*, pages 179–188, ACM, 2009.

[68] H. Mukhtar, D. Belaïd, and G. Bernard. User Preferences-based Automatic Device Selection for Multimedia User Tasks in Pervasive Environments. In *Proceedings of the Fifth International Conference on Networking and Services (ICNS)*, pages 43–48, IEEE, 2009.

[69] D. A. Norman. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution.* MIT Press, 1998.

[70] Oracle Inc. Home – Project Kenai. Retrieved May 2, 2015 from `http://visualvm.java.net/`.

[71] Oracle Inc. Java ME - The Most Ubiquitous Application Platform for Mobile Devices. Retrieved October 22, 2010 from `http://www.oracle.com/technetwork/java/javame/overview/index.html`.

[72] Oracle Inc. Java SE Overview - At a Glance. Retrieved October 22, 2010 from `http://www.oracle.com/technetwork/java/javase/overview/index.html`.

[73] Oracle Inc. MySQL - The World's Most Popular Open Source Database. Retrieved October 22, 2010 from `http://www.mysql.com/`.

[74] Oracle Inc. Overview (Connected Limited Device Configuration 1.1). Retrieved October 28, 2010 from `http://download.oracle.com/javame/config/cldc/ref-impl/cldc1.1/jsr139/index.html`.

[75] Oracle Inc. Overview (Java Platform SE 6). Retrieved October 22, 2010 from `http://download.oracle.com/javase/6/docs/api/index.html`.

[76] S. Outemzabet and C. Nerguizian. Accuracy Enhancement of an Indoor ANN-based Fingerprinting Location System Using Particle Filtering and a Low-Cost Sensor. In *Proceedings of the IEEE Vehicular Technology Conference (VTC)*, pages 2750–2754, IEEE, 2008.

[77] A. Padovitz, S. W. Loke, and A. Zaslavsky. Towards a Theory of Context Spaces. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 38–42, IEEE, 2004.

[78] J. M. Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward. Structured Decomposition of Adaptive Applications. *Pervasive and Mobile Computing*, volume 4, number 6, pages 791–806, 2008.

[79] J. Pascoe. The Stick-e Note Architecture: Extending the Interface Beyond the User. In *Proceedings of the Second International Conference on Intelligent User Interfaces (IUI)*, pages 261–264, ACM, 1997.

[80] P. Pawar, B.-J. van Beijnum, H. Mei, and H. Hermens. Towards Proactive Context-aware Service Selection in the Geographically Distributed Remote Patient Monitoring System. In *Proceedings of the Forth International Symposium on Wireless Pervasive Computing (ISWPC)*, pages 1–8, IEEE, 2009.

[81] J. Petzold. Augsburg Indoor Location Tracking Benchmark Data Set. Retrieved August 15, 2011 from `http://www.pervasive.jku.at/Research/Context_Database/`.

[82] J. Petzold. Augsburg Indoor Location Tracking Benchmarks. Technical report, Technisch-Naturwissenschaftliche Fakultät, Johannes Kepler Universität Linz, 2004.

[83] J. Petzold. *Zustandsprädiktoren zur Kontextvorhersage in ubiquitären Systemen.* PhD thesis, Fakultät für Angewandte Informatik, Universität Augsburg, 2005.

[84] J. Petzold, F. Bagci, W. Trumler, and T. Ungerer. Global and local state context prediction. *Artificial Intelligence in Mobile Systems*, 2003.

[85] T. Pitkäranta, O. Riva, and S. Toivonen. Designing and Implementing a System for the Provision of Proactive Context-aware Services. In *Proceedings of the Workshop on Context Awareness for Proactive Systems (CAPS)*, volume 1, pages 21–30, 2005.

[86] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, volume 9, number 3, pages 268–299, 1993.

[87] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A High-level Programming Model for Pervasive Computing Environments. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 7–16, IEEE, 2005.

[88] K. Rasch, F. Li, S. Sehic, R. Ayani, and S. Dustdar. Context-driven Personalized Service Discovery in Pervasive Environments. *World Wide Web*, volume 14, number 4, pages 295–319, 2011.

[89] J. Rekimoto, Y. Ayatsuka, and K. Hayashi. Augment-able Reality: Situated Communication through Physical and Digital Spaces. In *Proceedings of the Second International Symposium on Wearable Computers (ISWC)*, pages 68–75, IEEE, 1998.

[90] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck. Towards a Generic Observer/Controller Architecture for Organic Computing. *GI Jahrestagung (1)*, volume 93, pages 112–119, 2006.

[91] O. Riva, T. Pitkäranta, and S. Toivonen. Proactive Context-aware Service Provisioning for a Marine Community. In *Proceedings of the Workshop on Context Awareness for Proactive Systems (CAPS)*, pages 175–178, 2005.

[92] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, volume 1, number 4, pages 74–83, 2002.

[93] M. Román, F. Kon, and R. H. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: The dynamicTAO Case. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS Workshops)*, pages 122–127, IEEE, 1999.

[94] S. Russell and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall, 1995.

[95] N. S. Ryan, J. Pascoe, and D. R. Morse. Enhanced Reality Fieldwork: The Context Aware Archaeological Assistant. In *Archaeology in the Age of the Internet – Proceedings of the 25th Anniversary Conference on Computer Applications and Quantitative Methods in Archaeology (CAA)*, pages 269–274, 1998.

[96] S. M. Sadjadi and P. K. McKinley. A Survey of Adaptive Middleware. Technical Report MSU-CSE-03-35, Department of Computer Science and Engineering, Michigan State University, 2003.

[97] D. Salber, A. K. Dey, and G. D. Abowd. Ubiquitous Computing: Defining an HCI Research Agenda for an Emerging Interaction Paradigm. Technical Report GIT-GVU-98-01, College of Computing, Georgia Institute of Technology, 1998.

[98] A. Salovaara and A. Oulasvirta. Six Modes of Proactive Resource Management: A User-centric Typology for Proactive Behaviors. In *Proceedings of the third Nordic Conference on Human-Computer Interaction (NordiCHI)*, pages 57–60, ACM, 2004.

[99] I. Satoh. A Location Model for Pervasive Computing Environments. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 215–224, IEEE, 2005.

[100] M. Satyanarayanan. Accessing Information on Demand at Any Location – Mobile Information Access. *IEEE Personal Communications*, volume 3, number 1, pages 26–33, 1996.

[101] M. Satyanarayanan. Mobile Computing: Where's the Tofu? *ACM SIGMOBILE Mobile Computing and Communications Review*, volume 1, number 1, pages 17–21, 1997.

[102] M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, volume 20, number 2, pages 85–124, 2002.

[103] G. Schiele, C. Becker, and K. Rothermel. Energy-efficient Cluster-based Service Discovery for Ubiquitous Computing. In *Proceedings of the Eleventh ACM SIGOPS European Workshop (EW)*, page 14, ACM, 2004.

[104] B. Schilit, N. Adams, and R. Want. Context-aware Computing Applications. In *Proceedings of the First Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 85–90, IEEE, 1994.

[105] W. Schilit. *A System Architecture for Context-aware Mobile Computing*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995.

[106] A. Schmidt, M. Beigl, and H.-W. Gellersen. There is More to Context than Location. *Computers & Graphics*, volume 23, number 6, pages 893–901, 1999.

[107] A. Schmidt and K. Van Laerhoven. How to Build Smart Appliances? *IEEE Peronal Communications*, volume 8, number 4, pages 66–71, 2001.

[108] A. Schroeder, M. van der Zwaag, and M. Hammer. A Middleware Architecture for Human-centred Pervasive Adaptive Applications. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 138–143, IEEE, 2008.

[109] S. Sigg. *Development of a Novel Context Prediction Algorithm and Analysis of Context Prediction Schemes*. PhD thesis, Fachbereich Elektrotechnik/Informatik, Universität Kassel, 2008.

[110] S. Sigg, S. Haseloff, and K. David. An Alignment Approach for Context Prediction Tasks in Ubicomp Environments. *IEEE Pervasive Computing*, volume 9, number 4, pages 90–97, 2010.

[111] G. Simon, A. Lendasse, M. Cottrell, J.-C. Fort, and M. Verleysen. Time Series Forecasting: Obtaining Long Term Trends with Self-organizing Maps. *Pattern Recognition Letters*, volume 26, number 12, pages 1795–1808, 2005.

[112] J. P. Sousa and D. Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Software Architecture*, pages 29–43, Springer, 2002.

[113] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based Adaptation for Ubiquitous Computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, volume 36, number 3, pages 328–340, 2006.

[114] D. Tennenhouse. Proactive Computing. *Communications of the ACM*, volume 43, number 5, pages 43–50, 2000.

[115] A.-M. Vainio, M. Valtonen, and J. Vanhala. Proactive Fuzzy Control and Adaptation Methods for Smart Homes. *IEEE Intelligent Systems*, volume 23, number 2, pages 42–49, 2008.

[116] K. H. Wang and B. Li. Group Mobility and Partition Prediction in Wireless Ad-hoc Networks. In *Proceedings of the IEEE International Conference on Communications (ICC)*, volume 2, pages 1017–1021, IEEE, 2002.

[117] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, volume 10, number 1, pages 91–102, 1992.

[118] R. Want, T. Pering, and D. Tennenhouse. Comparing Autonomic and Proactive Computing. *IBM Systems Journal*, volume 42, number 1, pages 129–135, 2003.

[119] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, volume 4, number 5, pages 42–47, 1997.

[120] M. Weiser. The Computer for the 21st Century. *Scientific American*, volume 265, number 3, pages 94–104, 1991.

[121] P. Whitle. *Hypothesis Testing in Time Series Analysis*, volume 4, Almqvist & Wiksells, 1951.

[122] O. Wolfson, L. Jiang, A. P. Sistla, M. Deng, S. Chamberlain, and N. Rishe. Databases for Tracking Mobile Units in Real Time. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 169–186, Springer, 1999.

Publications Contained in This Thesis

- S. VanSyckel and C. Becker. A Survey of Proactive Pervasive Computing. In *Proceedings of the 3rd Workshop on Recent Advances in Behavior Prediction and Pro-active Pervasive Computing (AwareCast)* in conjunction with the *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 421–430, ACM, 2014.
- S. VanSyckel, D. Schäfer, V. Majuntke, C. Krupitzer, G. Schiele, and C. Becker. COMITY: A Framework for Adaptation Coordination in Multi-Platform Pervasive Systems. *Pervasive and Mobile Computing*, volume 10, pages 51–65, 2014.
- S. VanSyckel, D. Schäfer, G. Schiele, and C. Becker. Configuration Management for Proactive Adaptation in Pervasive Environments. In *Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 131–140, IEEE, 2013.
- S. VanSyckel. Middleware-based System Support for Proactive Adaptation in Pervasive Environments. In *Proceedings of the Sixth Annual Ph.D. Forum on Pervasive Computing and Communications* at the *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 425–426, IEEE, 2013.
- S. VanSyckel, G. Schiele, and C. Becker. Extending Context Management for Proactive Adaptation in Pervasive Environments. In *Proceedings of the 7th International Conference on Ubiquitous Information Technologies and Applications (CUTE)*, pages 823–831, Springer, 2012.
- S. VanSyckel, G. Schiele, and C. Becker. POSTER: Towards Proactive Adaptation in Pervasive Environments. In *Proceedings of the 8th International ICST Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pages 214–218, Springer, 2011.

Lebenslauf

Seit 11/2012	Lehrbeauftragter DHBW Mannheim
Seit 12/2010	Akademischer Mitarbeiter Lehrstuhl für Wirtschaftsinformatik II Universität Mannheim
10/2005 – 11/2010	Diplom-Wirtschaftsinformatik Universität Mannheim
08/2008 – 05/2009	Informatik University of Massachusetts at Amherst