Towards an Enterprise Software Component Ontology

Stefan Seedorf
University of Mannheim
seedorf@uni-mannheim.de

Martin Schader
University of Mannheim
martin.schader@uni-mannheim.de

ABSTRACT

The paper describes an ontology of enterprise software components (ESCO). Its purpose is to reduce the conceptual gap between component specifications and their corresponding business descriptions, i.e., business processes and business entities. To this end, the different types of enterprise components and their semantic relationships are analyzed. The ontology axioms are formally expressed in description logics. ESCO contributes to a shared understanding of key concepts in application system development.

Keywords

Enterprise component, business process, ontology, component model, software architecture.

INTRODUCTION

Traditionally, there has been a conceptual gap between application software systems and the business domain. While application software systems are shaped by business processes, rules, and concepts, it is often not known how software components relate to business processes and vice versa. This is mainly due to the separation between the business and software life cycles. Application system development is concerned with the question of understanding and implementing the business domain, but often it is not explicitly traced how business processes and concepts are realized by the underlying components. In recent years, service-orientation has led to a more process-oriented perspective, but the conceptual gap has yet to be closed. An important motivation for closing this gap is to decrease architecture and design erosion (Bosch, 2004). As the business environment evolves over time, the software systems have to be aligned. However, the knowledge of how business processes and software are connected is often lost or incomplete, e.g., due to employee fluctuation or lack of documentation. As a result, the accumulated maintenance cost can exceed the initial development cost (Sommerville, 2007).

Standard modeling languages such as the UML provide the means for enterprise modeling, system analysis, and design. However, they have not been designed to create an integrated view and describe the semantic relationships (although this is of course possible). Moreover, business experts and software developers often use completely different modeling and specification languages. What is needed therefore is a common language for both describing the building blocks of software systems, i.e., components with business functionality, as well as the building blocks of the business domain together with the semantic relationships between them.

In this paper, we introduce an enterprise software component ontology (ESCO). The purpose of the ontology is to establish a shared understanding of enterprise software components, what types of components exist, and what their relationships to entities in the business domain are. Figure 1 shows the scope of our ontology. Both concepts from the software and business domain are included. The instances of ESCO are modeling abstractions during development time but not run-time or physical entities. The ontology is expressed in description logics, a knowledge representation language for describing terminologies and reasoning about them (Baader, 2003). It allows for high degrees of expressiveness, which is useful for a precise definition of the concepts, and in most cases offers decidability for automatic reasoning and discovering inconsistencies. The purpose of ESCO is not to replace existing modeling and specification languages but work towards a more standardized set of concepts for the development of application systems. To this end, the ontology lays the conceptual foundation for a better understanding of enterprise components.

The remainder of the paper is structured as follows: In the second section, we describe related work in the form of existing conceptual models. In the third section, the methodology used to develop the ontology is described. In section four, the concepts, relationships, and axioms of the ontology are described. Subsequently, the potential benefits and shortcomings of ESCO are discussed. In the conclusion, we summarize our findings and give directions for future work.

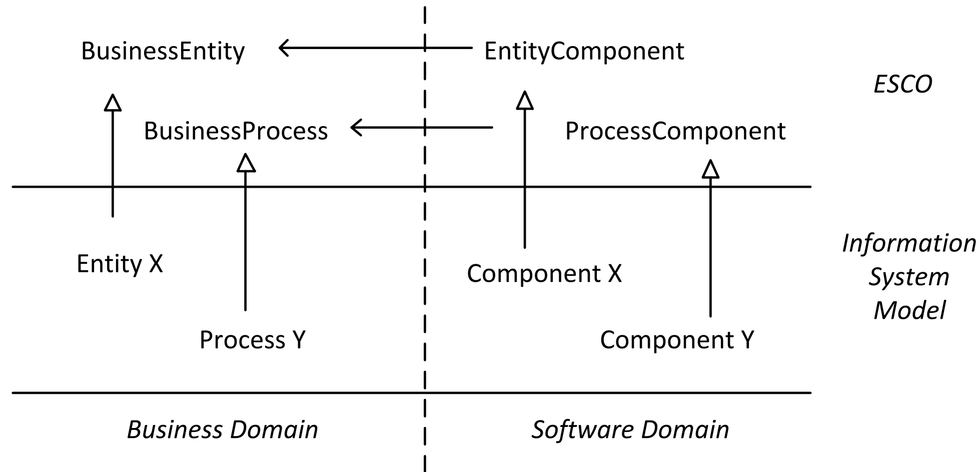

Figure 1. Ontology spanning both software and business concepts

RELATED WORK

Oberle (2006) formalizes various concepts in component-based and software-oriented development. He describes a "core ontology of software components" which extends a "core software ontology." All concepts are specified in modal logic and based on concepts in the foundational DOLCE ontologies (Masolo et al., 2002). According to Oberle, a software component is a class, which conforms to a framework specification. This interpretation is closely related to the Java Enterprise Edition, where a component is based on a Java class. This is sufficient for the described application scenario of a "semantic application server" but not generic enough for other languages that might use different constructs. Additionally, the ontology does not describe software components and their relationships to concepts from the enterprise domain.

Semi-formal models describe selected aspects of enterprise software components. The conceptual model introduced by Fettke et al. (2004) describes seven layers of a component specification. It includes a terminology layer which relates business terms to data types and services of a component specification. Korthaus (2000) describes a conceptual model as part of a component-based development method. He introduces three types of enterprise software components. However, his model does not describe their relations to concepts in the business domain.

In recent years, the vision of "semantic web services" has led to the emergence of several service ontologies. Although these ontologies build on the concepts of a "service" or "software service" and not "software component," they can be seen as related work as they not only describe software entities but also processes and, to some extent, business entities. OWL-S is a core ontology for semantic web services (Martin et al., 2006). The OWL-S service model decomposes a service definition into a workflow description, which consists of atomic and composite processes. Central OWL-S concepts, e.g., service and process, are not defined by the ontology. Nevertheless, it is useful to identify the semantic relationships between processes, services, and operations. The problem that there is no unified, ontological definition of "service" is addressed by Ferrario and Guarino (2009).

Semantic web services mainly focus on the software entities, by providing semantic descriptions of computational services so that they can be automatically discovered, orchestrated, and executed. There have been efforts to bridge the gap between semantic web services and business process management (BPM). The SUPER research project aims at integrating the business expert perspective on processes and services with the software perspective. Within this project, a number of process ontologies are proposed based on an "upper-level process ontology" that is derived from DOLCE and semantic web service ontology WSMO (Hepp and Roman, 2007).

METHODOLOGY

In the context of information systems, the term "ontology" refers to "the shared understanding of some domain of interest" (Uschold and Grüninger, 1996). If an ontology is described by axioms in a logical language such as description logics, the intended meaning of terms can be specified in more detail. In our research, we interpret an ontology as an artifact and thus follow a design science approach.

In the beginning, the ontology's purpose was stated as motivating scenario: "[…] *to describe a shared understanding of enterprise software components which integrates the perspective of business experts.*" Subsequently, six informal competency questions were specified. Competency questions are questions that must be answered by the ontology. Thus, they provide both an evaluation and stopping criteria. The following competency questions were specified:

(1) What are software components and basic elements of a component specification?

(2) Which relationships between software components exist?

(3) How are software components and services related?

(4) What is an enterprise component and how does it differ from a software component?

(5) What are common types of enterprise components?

(6) How are enterprise components related to concepts of the business domain, e.g. business processes?

In the next step, we answered the competency questions iteratively, starting with the definition of software components and enterprise components. Definitions were retrieved from the literature and subsequently analyzed. This was followed by a qualitative analysis of selected sources: the component-based process models SELECT perspective (Allen and Frost, 1998) and KobrA (Atkinson et al., 2001); three component models Enterprise JavaBeans (Sun Microsystems, 2006), CORBA Component Model (OMG, 2006), and Microsoft .NET (Microsoft, 2007); followed by the ontologies and meta models stated in the previous section. An unequivocal definition of enterprise and software component was not possible due to an either more technical or more business-oriented interpretation; however, we identified generic types of enterprise components and elements of component specification. The results were then written down in natural language and modeled by means of UML class diagrams. In the formalization stage, the concepts, relations, and axioms were then defined in description logics.

After formalization, a simple evaluation was performed by answering the competency questions. Moreover, the classification hierarchy was analyzed with the OntoClean method (Guarino and Welty 2002), which allows for a systematic analysis of classification hierarchies based on four meta-properties (identity, rigidity, dependence, unity). The ontological analysis led to the refinement of a super-subclass relationship.

Finally, the ontology was implemented in the Web Ontology Language (OWL) 2.0 with minimal adjustments. The class hierarchy was automatically tested with Protégé and Pellet Reasoner. Instance reasoning was tested with a case example from the private banking domain, derived from a simplified but fully functional Java implementation. Due to space limitations, the example has not been included in this paper.

DESCRIPTION OF THE ONTOLOGY

First, the concept software component is defined and two kinds of enterprise component are identified. Second, concepts from the enterprise domain are characterized. In the last part, we bridge the gap between the software and business view by defining ontological relationships between the concepts.

Enterprise and Software Component

The term "component" is widely used in IS academia and practice but includes various interpretations. Further, there are different kinds of component, i.e., software components, enterprise (or business) components, and business objects whose interpretations differ as well. The main goal of components is that they can be reused in different contexts. Following this school of thought going back to the 1968 NATO software engineering conference (Naur & Randell, 1969), there is an abundance of interpretations of the term. Any software artifact that can be reused is seen as a component. First, such interpretation comprises routines, functions, modules, classes, and modules. Second, components can be not only code but any artifact suitable for reuse, e.g., design patterns, requirements, documentation of domain knowledge, etc. (Sametinger, 1997). On the other side, narrow interpretations are restricted to binary software artifacts with well-defined interfaces.

It is important to differentiate between components at development-time and components at run-time. The first are "software components" while the latter are "system components." A software component may serve as a blueprint, e.g., a class, or a system component, e.g., an object. A widely accepted definition of software components is the following: "A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties" (Szyperski et al., 2002).

However, it is possible that a component does not have an interface and externally specified dependencies to be composed. A more general definition is that a software component is an independent software unit, which can be composed with other

components to create a software system (Sommerville, 2007). In order to be deployable in some other context without or with only little changes, a software component has to conform to the requirements imposed by a software architecture (Heineman & Councill, 2001). This prerequisite is usually fulfilled by component models and technologies, which can be part of a programming language or an extension of it. Provided that a software component is understood as a piece of software (and not an artifact such as a design pattern), it is inseparably connected to a component model.

According to this understanding, we define a software component as a unit of software that conforms to some component model:

$$\mathtt{SoftwareComponent} \equiv \mathtt{Software} \sqcap \exists \mathtt{conforms.ComponentModel}$$

This definition does not impose any constraints on the component model. A simple example of a component model is the JavaBeans specification (Hamilton, 1997). A JavaBean is a Java class that is designed in a way that is suitable for reuse, and thus it is a component. Also a .NET assembly, which is a container for one or more binaries, can be seen as a component (Microsoft, 2007).

Software components have a specification and one or more realizations (Sommerville, 2007). The most common type of specification is an interface specification describing the static contract of a component (see Figure 2).

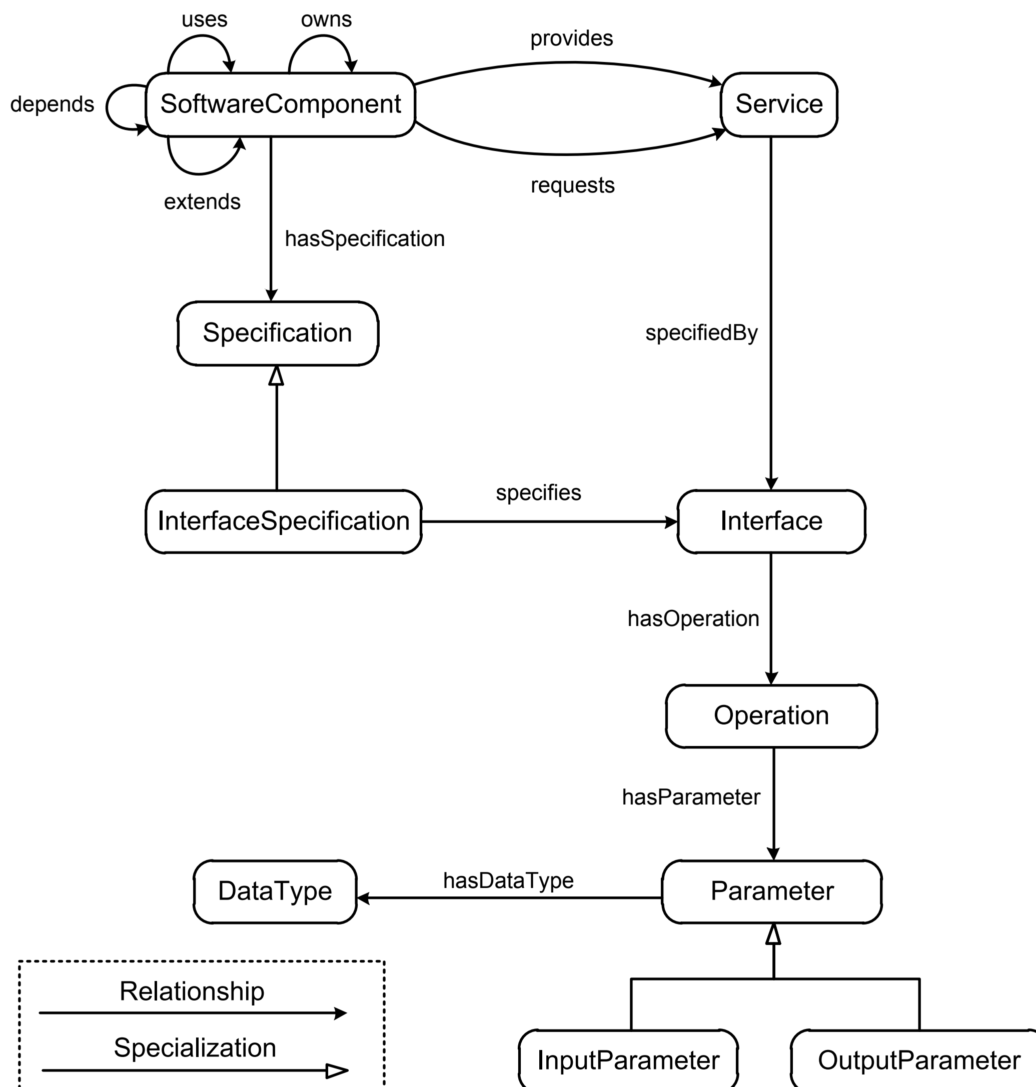

Figure 2. Software component

A component can have an arbitrary number of provided and required interfaces. If a component specifies a required interface this usually means that it contractually requires another software component. This creates a dependency between components, which is described by the relationship “depends.” If component A depends on component B, and component B depends on C, then A also depends on C. Therefore, the relationship “depends” is transitive.

Every interface can be broken down into operations that have input and output parameters. Every parameter has a data type such as String or a complex data type. Please note that the concepts and relationships do not replace a full interface specification of a component, instead they represent a higher level of abstraction leaving out details not needed for the ontological relationships.

While the functionality of a component can be specified using one to many interfaces, one can also say that a component provides and requests “services.” Services are abstractions of the functionality provided or requested by a component. However, not all components provide services.

Herzum and Sims (2000) state that enterprise components realize business processes or business entities. While the meta model by Herzum and Sims defines four types of enterprise components, most component models describe at least two types: entity and process components (see Figure 3):

$$\text{ProcessComponent} \sqcup \text{EntityComponent} \sqsubseteq \text{EnterpriseComponent}$$

Enterprise JavaBeans (EJB) and the CORBA component models define several component types that either represent process or entity components (OMG, 2006; Sun Microsystems, 2006). The .NET component model is generic but both component types can be realized by using additional APIs (Microsoft, 2004). Please note that this classification hierarchy is still extensible with more types as suggested by Herzum and Sims (2000).

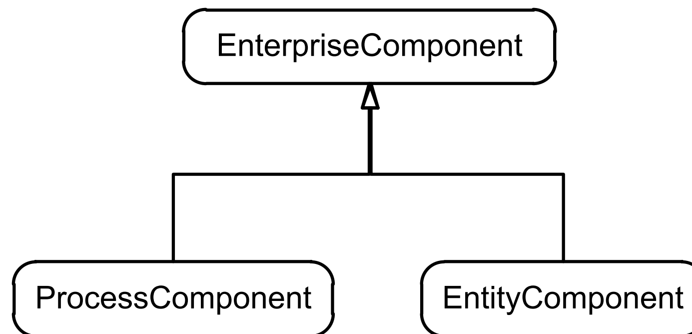

Figure 3. Enterprise component

As stated before, the term “business object” is ambiguous and may be used by business experts and software developers to describe different concepts. Therefore, we suggest using either “enterprise component” or “domain entity.” The latter is explained in the following section.

Business Processes and Business Entities

Business processes are an integral part of any information system. Davenport (1992) defines a business process as “a structured, measured set of activities designed to produce a specific output for a particular customer or market.” Here, we define a business process as a process that has a business domain as its domain:

$$\text{BusinessProcess} \equiv \text{Process} \sqcap \exists \text{hasDomain.BusinessDomain}$$

This interpretation would not contradict a more complex definition of business processes, e.g., based on activities and outputs. However, it is not our purpose to create a formal enterprise ontology (cf. Fox & Gruninger, 1998). Instead, we limit the scope of our research to elements that are needed for closing the conceptual gap between the software and business domains. In Davenport’s and many others’ definitions, business processes consist of a set of activities or process steps. Each process can be broken down into smaller steps, which are either atomic processes or composite processes that can be further broken down into even smaller parts. The atomic processes are called “tasks,” while the process steps that can be further broken down are called “subprocesses” (see Figure 4). Processes and process steps have inputs and outputs. The process steps are executed according to some process logic which is not part of the model.

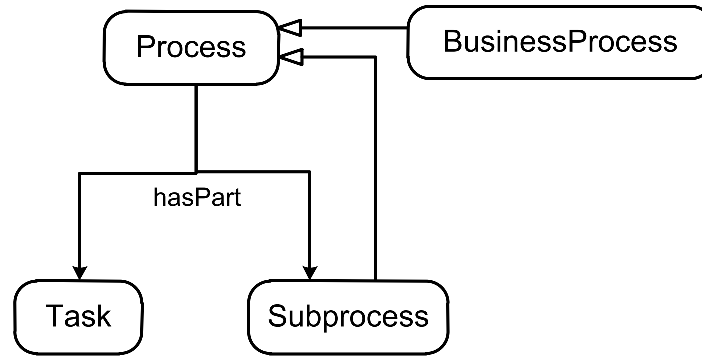

Figure 4. Process and business process

Other parts of any enterprise architecture are the "business entities" or "domain entities." Business entities are often realized in a software system, e.g., as entity components or data types. However, it is clear how the components relate to business entities. Business entities can be specified in different ways: a glossary, a thesaurus, an entity relationship model, an UML class diagram, or a domain ontology. For traceability between enterprise and software architecture, the following relationships between business entities are relevant:

- Synonymous relationship (sameAs)
- Similarity relationship (similarTo)
- Super-/subconcept

The first two are relationships typically specified in a thesaurus. In order to find components that realize a concept it is often useful to include the same and similar concepts in the search. The relationships "similarTo" and "sameAs" are symmetric. The "sameAs" relationship is also transitive. The super-/subconcept relationship states that one concept is included in another, e.g., "checking account" is a subconcept of "account." This is possible by describing these relationships between business entities. Additional domain-specific relationships may be defined in an ER model or UML class diagram but they are not necessary for traceability.

Ontological Relationships

In this section, we describe the ontological relationships between enterprise components, business processes, and business entities. Usually, process components not fully implement a business process but only realize single steps of a process. The reason is that software systems are generally broken down into smaller parts by applying principles of low coupling and high cohesion. Further, supporting a business process often spans across several software systems.

A process component is an enterprise component, which supports some business process:

$$\text{ProcessComponent} \equiv \text{EnterpriseComponent} \sqcap \exists \text{supportsProcess.BusinessProcess}$$

Thus, a process component is defined by the process(es) it supports (see Figure 5). This basic ontological relationship is important because it relates to both the software and the business domains. When describing this relationship across software systems, business processes can be traced to business processes and vice-versa. For example, such knowledge is useful in a change impact analysis where one needs to find all components that could be affected by the change of a business process. In many cases, such knowledge is usually not readily available but must be extracted from documentation by performing a local search.

Further, we previously introduced another type of enterprise components that do not directly support a business process. These are the entity components, which model persistent objects of the business domain:

$$\text{EntityComponent} \equiv \text{EnterpriseComponent} \sqcap \exists \text{realizesConcept.BusinessEntity}$$

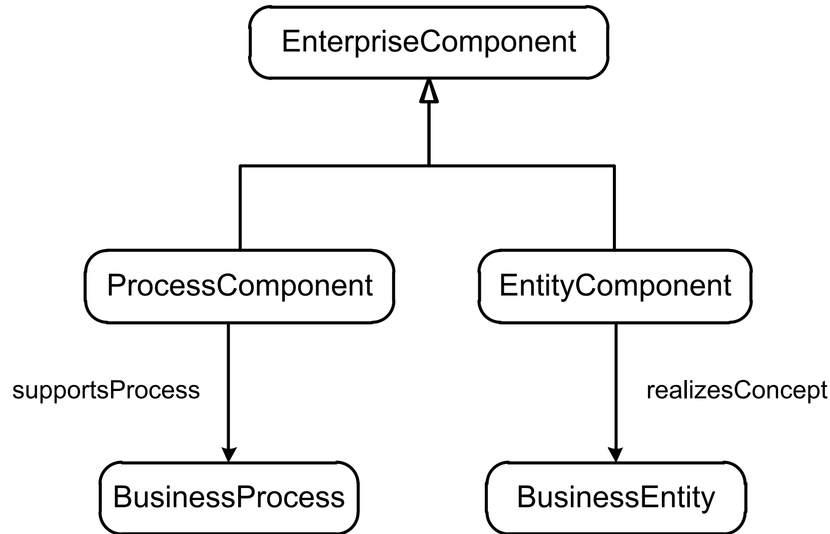

Figure 5. Ontological relationships of process and entity component

In most cases, entity components realize exactly one concept but sometimes they may realize more than one. This relationship is useful to find components according to the business entities or a similar concept they realize. It also helps to understand what a component is about. In addition to its application to entity components, this relationship is used to define data types (see Figure 6).

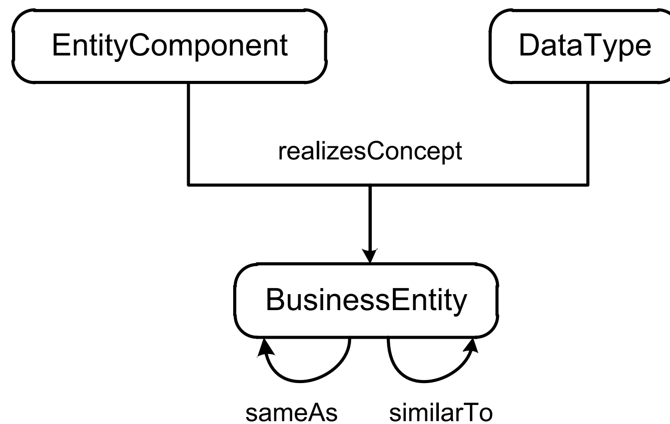

Figure 6. Relationships between entity component, data type, and business entity

In the case of process components, the ontological relationships can be specified at the operation level. The operation of an interface can support a task (process step) which is part of a process. This kind of relationship has been described in the OWL-S ontology in a similar way (Martin *et al.*, 2006). In OWL-S, atomic processes (tasks) are mapped to operations defined by services. However, the OWL-S ontology relates executable processes to services instead of business processes to components. The relationship "supportsTask" describes that an operation supports a process task (see Figure 7).

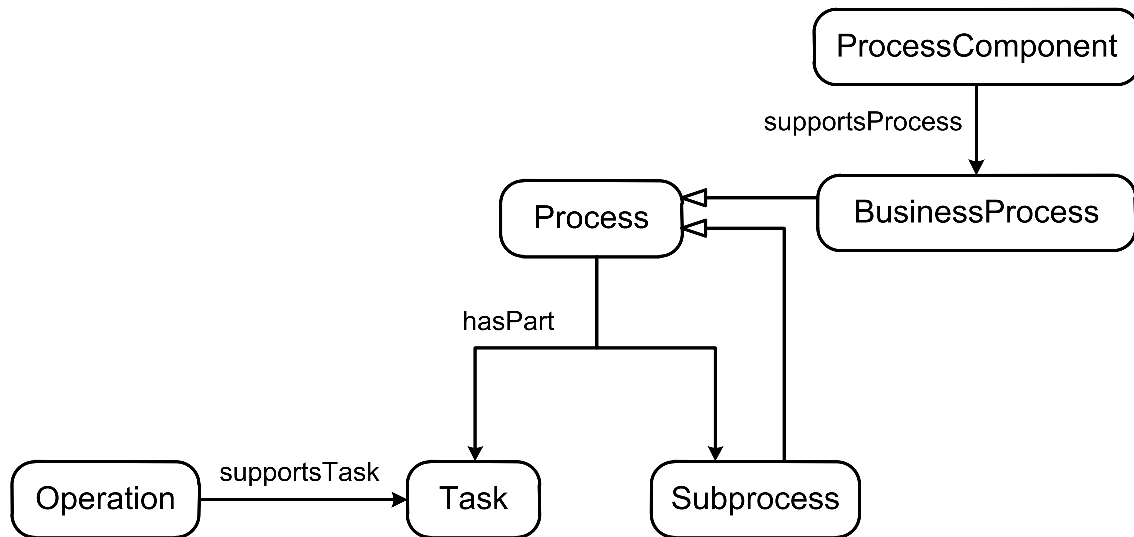

Figure 7. Relationships between operation, task, and process

It is assumed that a component providing a service specified by an interface operation, which supports task T, where T is part of the process P, must also support process P:

$$\text{provides} \circ \text{specifiedBy} \circ \text{hasOperation} \circ \text{supportsTask} \circ \text{partOf} \rightarrow \text{supportsProcess}$$

The above axiom is very useful when we create a knowledge base that contains process component instances. If an interface operation is annotated with a task, it can automatically be expected that the component providing this operation supports the associated business process. Subsequently, a reasoner can automatically infer that this component is a process component. That is why using description logic as a formalism to describe ontological relationships has some practical advantages compared to describing the relationships in an ER/database approach or using UML/MOF that do not support automated reasoning on a terminology or knowledge base.

DISCUSSION

The ontology described above has been formalized in description logics (see Appendix). As stated before, a rich axiomatization allows for a precise definition of concepts. Because ESCO is both a conceptual model and an implementation artifact, it can be used in two ways:

- Reference model: This relates not to the descriptive but to the normative character of ontologies. ESCO promotes an unequivocal interpretation of ambiguous concepts such as “software component.” The model supports business experts and software developers to become more conscious of the relationships between software systems and the business domain. However, it should also be noted that the description logics notation does not promote practical use. Nevertheless, the ontology can serve as a reference for the development of simplified models.
- Application system knowledge base: ESCO was implemented in OWL, which enables the creation of an integrated, abstract view of software systems during development time. In this case, software component and business process descriptions are managed in a knowledge base. These descriptions may furthermore be automatically extracted from software artifacts, e.g., UML models and Java source code (Happel and Seedorf, 2007, Seedorf 2010). A software information system based on ESCO may be applied to traceability management, governance, and quality control.

ESCO is currently limited to the definition of software and enterprise components. Although the component concepts are derived from more general ones, other concepts such as software and service are atomic concepts in the ontology. This can be resolved by building these concepts on foundational ontologies, e.g., DOLCE, as it has been studied elsewhere (see section 2). An extension of the ontology should also cover the notion of run-time concepts such as system component, process instance, and information object.

Despite the fact that ESCO only describes development-time concepts, they might still be mistaken for run-time concepts. For example, “SavingsAccount” is an instance of enterprise component in ESCO. The instance, however, is the definition of a software component, e.g., a Java class, and not the object “12AB54F:SavingsAccount” at run-time. The dependencies between components are specified at development-time, but they may come into being at compile or run-time. Further, the physical entity “Bob” may be both represented by an information object during system analysis (e.g., in an UML object diagram) as well as by a computational object during system run-time. The ontology, however, deals with none of those objects. It only looks at the modeling abstractions, e.g., the UML classes “Person” and “Employee”, and the software abstractions, e.g., the Java class “Employee.” The ontological relationships also deal with the abstractions in the business and software domain.

Another possible issue may be caused by the concept “BusinessEntity.” While on the ontology level this is clearly a concept, every instance can be both an instance of “BusinessEntity” and a concept. For example, the ontology allows that “Employee” is defined as a sub-concept of “Person.” Now if “Person” and “Employee” are instances of “BusinessEntity” they assume both roles. At first glance, this causes a contradiction. However, such cases are often found in conceptual modeling. For a comprehensive discussion of this problem and the implications for ontology reasoning see Motik (2007). The application system knowledge base was implemented as a functional prototype and instantiated with a case example. As our focus is on the basic concepts, neither the implementation nor the example are discussed in this paper.

CONCLUSION

In recent years, the service paradigm has contributed to a better alignment of software systems to business processes. However, from the perspective of software components there remains a conceptual gap between the software and business domain. The proposed ontology provides a first step towards a shared understanding of “enterprise component” and its semantic relationships. By defining this concept in a logical language, it was possible to specify their intended meaning explicitly. There are two benefits of this: First, ESCO can improve communication between business experts and software developers if the ontology serves as a reference model. Second, the ontology can be implemented as a knowledge base for application software systems.

The proposed ontology is not yet completed. First, only a few of the concepts are derived from more generic ones. This can be resolved by grounding all concepts on a foundational ontology such as DOLCE. As description logics is based on the open-world assumption, the ontology can be extended without falsifying the encoded axioms. Second, more concepts, e.g., application software system, goal, and requirement, may be included in the future.

ESCO does not substitute existing modeling languages because it lacks an appropriate graphical notation and corresponding tools. On the other hand, the implementation of ESCO allows for integrating and reasoning about software and business knowledge. If presented in a user-friendly way, the knowledge of both business experts and software experts may be enhanced without replacing their existing tools and modeling languages.

REFERENCES

1. Allen, P. and Frost, S. (1998) Component-Based Development for Enterprise Systems: Applying the SELECT Perspective: Cambridge University Press.
2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R. et al. (2001) Component-Based Product Line Engineering with UML. Addison-Wesley Professional.
3. Baader, F. (2003) The description logic handbook: Theory, implementation, and applications: Cambridge Univ Press.
4. Barbier, F. (2003) Business component-based software engineering: Springer.
5. Bosch, J. (2004) Software architecture: The next step. Lecture Notes in Computer Science, 3047, 194-199.
6. Davenport, T. H. (1992) Process innovation: Reengineering work through information technology: Harvard Business Press.
7. Fellner, K., Rautenstrauch, C. and Turowski, K. (1999) Fachkomponenten zur Gestaltung betrieblicher Anwendungssysteme. IM Information Management & Consulting, 14(2), 25-34.
8. Ferrario R., and Guarino, N. (2009) Towards an Ontological Foundation for Services Science, Proceedings of Future Internet Symposium 2008, Springer Verlag, Lecture Notes in Computer Science, Vol. 5468, 152-169.
9. Fettke, P., Loos, P. and Pastor, K. (2004) Ein UML-basiertes Metamodell zum Memorandum zur vereinheitlichten Spezifikation von Fachkomponenten. In: K. Turowski, AKA, 181-201.

10. Fox, M. S., and Gruninger, M. (1998) Enterprise modeling. AI Magazine, 19(3), 109.
11. Guarino, N. and Welty, C. (2002) Evaluating Ontological Decisions with OntoClean. Communications of the ACM, 45(2), 61-65.
12. Hamilton, G. (1997) JavaBeans specification: Sun Microsystems Inc.
13. Happel, H. J. and Seedorf, S. (2007) Ontobrowse: A semantic wiki for sharing knowledge about software architectures. Proceedings of the 19th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE), Boston, USA.
14. Heineman, G. T. and Councill, W. T. (2001) Component-based software engineering: Putting the pieces together: Addison-Wesley Professional.
15. Hepp, M. and Roman, D. (2007) An Ontology Framework for Semantic Business Process Management, Proceedings of Wirtschaftsinformatik.
16. Herzum, P. and Sims, O. (2000) Business component factory: A comprehensive overview of component-based development for the enterprise. Wiley, New York.
17. Korthaus, A. (2001) Komponentenbasierte Entwicklung computergestützter betrieblicher Informationssysteme. Peter Lang Verlag, Frankfurt am Main.
18. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., et al. (2006) Owl-s: Semantic markup for web services. Version 1.2. http://www.ai.sri.com/daml/services/owl-s/1.2/overview/
19. Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A. and Schneider, L. (2002): The WonderWeb Library of Foundational Ontologies. WonderWeb Deliverable, 17.
20. Microsoft (2004) Anwendungsarchitektur für .NET Entwerfen von Anwendungen und Diensten. http://msdn.microsoft.com/de-de/library/bb979301.aspx
21. Microsoft (2007) MSDN library –.NET framework 3.5. http://msdn.microsoft.com/en-us/library/w0x726c2.aspx
22. Motik, B. (2007) On the Properties of Metamodeling in OWL. Journal of Logic Computation, 17(4), ISSN 0955-792X, 617-637.
23. Naur, P., & Randell, B. (1969) Software engineering: Report on a conference sponsored by the Nato Science Committee, Garmisch, Germany, 7th to 11th October 1968.
24. Oberle, D. (2006) Semantic Management of Middleware. Springer, New York; Heidelberg.
25. OMG (2006) CORBA component model 4.0 specification. http://www.omg.org/docs/formal/06-04-01.pdf
26. Sametinger, J. (1997) Software engineering with reusable components. Springer, Berlin; Heidelberg.
27. Seedorf, S. (2010): Ontologie-gestützte Entwicklung komponentenbasierter Anwendungssysteme. Peter Lang Verlag, Frankfurt a.M.
28. Sims, O. 1994. Business objects: Delivering cooperative objects for client-server: McGraw-Hill, Inc. New York, NY, USA.
29. Sommerville, I. (2007) Software engineering (8 ed.). Addison-Wesley, Harlow; München.
30. Sun Microsystems (2006) JSR 220: Enterprise JavaBeans, Version 3.0: Java Community Process.
31. Szyperski, C., Gruntz, D. W. and Murer, S. (2002) Component software. Addison-Wesley, London.
32. Uschold, M. and Gruninger, M. 1996: Ontologies: Principles, Methods, and Applications. Knowledge Engineering Review, 11(2), 93-155.
33. Wang, Z., Xu, X. and Zhan, D. (2006). A survey of business component identification methods and related techniques. International Journal of Information Technology, 4(2), 229-238.

APPENDIX

The following listing includes all ESCO axioms in description logics:

```
01 SoftwareComponent ≡ Software ⊓ ∃ conforms.ComponentModel
02 SoftwareComponent ⊓ ∃ provides.Service ⊑ ServerSoftware
03 SoftwareComponent ⊓ ∃ requests.Service ⊑ ClientSoftware
04 ServerSoftware ⊔ ClientSoftware ⊑ Software
05 InterfaceSpecification ⊑ Specification
06 ProvidedInterface ⊑ Interface
07 RequiredInterface ⊑ Interface
08 ∃ specifies(∃ provides.Service)⊑ ProvidedInterface
09 ∃ specifies(∃ requests.Service)⊑ RequestedInterface
10 InputParameter ⊑ Parameter
11 OutputParameter ⊑ Parameter
12 Transitive(depends)
13 ⩾1 conforms ⊑ Software
14 ⊤ ⊑ ∀ conforms.ComponentModel
15 ⩾1 provides ⊑ SoftwareComponent
16 ⊤ ⊑ ∀ provides.Service
17 ⩾1 requests ⊑ SoftwareComponent
18 ⊤ ⊑ ∀ requests.Service
19 ⩾1 hasSpecification ⊑ Software
20 ⊤ ⊑ ∀ hasSpecification.Specification
21 ⩾1 constitutedBy ⊑ Specification
22 ⩾1 specifiedBy ⊑ Service ⊔ InterfaceSpecification
23 ⊤ ⊑ ∀ specifiedBy.Interface
24 ⩾1 hasOperation ⊑ Interface
25 ⊤ ⊑ ∀ hasOperation.Operation
26 ⩾1 hasParameter ⊑ Operation
27 ⊤ ⊑ ∀ hasParameter.Parameter
28 ⩾1 hasDataType ⊑ Parameter
29 ⊤ ⊑ ∀ hasDataType.DataType
30 requestedBy ≡ requests¯
31 providedBy ≡ provides¯
32 specifies  ≡ specifiedBy¯
33 EnterpriseComponent ≡ SoftwareComponent ⊓ ∃ hasDomain.BusinessDomain
34 ProcessComponent ≡ EnterpriseComponent ⊓ ∃ supportsProcess.BusinessProcess
35 EntityComponent ≡ EnterpriseComponent ⊓ ∃ realizesConcept.BusinessEntity
36 ⊤ ⊑ ∀ hasDomain.BusinessDomain
37 ⊤ ⊑ ∀ realizesConcept.BusinessEntity
38 ⩾1 supportsProcess ⊑ ProcessComponent
39 BusinessProcess  ≡ Process ⊓ ∃ hasDomain.BusinessDomain
```

```
40 ⊤ ⊑ ∀ hasPart.(SubProcess ⊔ Task)
41 partOf  ≡ hasPart⁻
42 ⊤ ⊑ ∀ supportsTask.Task
43 ⊤ ⊑ ∀ supportsProcess.BusinessProcess ⊔ Task
44 ⩾1 realizesConcept ⊑ EntityComponent ⊔ DataType
45 ⩾1 similarTo ⊑ BusinessEntity
46 ⊤ ⊑ ∀ similarTo.BusinessEntity
47 Symmetric(similarTo)
48 ⩾1 sameAs ⊑ BusinessEntity
49 ⊤ ⊑ ∀ sameAs.BusinessEntity
50 Transitive(sameAs)
51 Symmetric(sameAs)
52 similarTo ○ sameAs → similarTo
53 sameAs ○ similarTo → similarTo
54 provides ○ specifiedBy ○ hasOperation ○ supportsTask  ○ partOf → supportsProcess
```