Round-Trip Engineering of Ontologies for Knowledge-Based Systems

Holger Knublauch and Thomas Rose

Research Institute for Applied Knowledge Processing (FAW), Ulm, Germany
E-mail: {Holger.Knublauch|Thomas.Rose}@faw.uni-ulm.de

Abstract

Knowledge Engineering methodologies suggest to develop knowledge-based systems by means of abstract conceptual models such as ontologies. However, they provide little support for integrating these models into the overall software architecture. As a result, moving from high-level conceptual models to a reliable and maintainable implementation is difficult and expensive. The central idea of the Software Engineering framework for knowledge-based systems presented in this paper is to amalgamate ontology construction with an object-oriented development process. The ontologies, designed as UML class diagrams with OCL constraints, are automatically mapped to structure-preserving Java classes. Meta-information on these classes, which is extracted at run-time by means of reflection, enables the reuse of generic components for knowledge acquisition, consistency checking, schema translation and knowledge sharing. The approach enables the efficient round-trip engineering of knowledge-based systems by means of smoothly integrating ontologies into the software architecture, massive tool support, compatibility with existing standards and rapid prototyping.

1. Introduction

There is an increasing demand for "intelligent" software in domains as diverse as clinical decision making, business process construction and intelligent tutoring. In contrast to other types of software, like text processors and database applications, these systems contain an implicit or explicit model of domain knowledge. The elicitation and maintenance of this domain knowledge in programs is difficult since knowledge is often hard to formalize and rarely well-understood by computer scientists. Thus, domain experts must be involved in system design and evaluation. Ideally, the system's knowledge should be made transparent, so that the domain experts can understand, edit and evaluate the knowledge. This is especially important in safety-critical decision-support systems, which rely on a traceable transformation of requirements and knowledge into the executable system [14].

Responding to this need for high-quality systems, researchers from the field of Knowledge Engineering (KE) [28] have proposed systematic guidelines and techniques for the construction of knowledge-based systems (KBSs), such as CommonKADS [27] and Protégé [20]. At the core of these methodologies is the construction of explicit models of domain expertise, consisting of knowledge about the domain concepts (ontologies) [33], problem-solving methods (PSMs) and mapping rules that specify how to apply PSMs to domain knowledge [23].

However, although being matters of spirited discussion on an academic level for more than a decade, those theoretical advances still await their breakthrough in industrial projects [2]. We argue that this is largely because until now little effort has been made to integrate knowledge models into the overall software architecture. Ontologies are represented in languages derived from KIF [11] or KL-ONE [5], which are little known to developers outside the field of Artificial Intelligence (AI). Furthermore, these languages make it hard to implement interactions between the knowledge-representing components and the remaining modules such as GUIs, which are typically designed and implemented in object-oriented languages using commercial tools. Finally, although tool support is vital for efficient system development, there are hardly any ontology modeling tools of industrial strength available. As a result, moving from high-level conceptual models to a reliable and maintainable implementation is difficult [2] and developers have to expend considerable efforts building up a basic infrastructure to embed KE technology into their projects.

In order to overcome these limitations of a pure AI perspective on developing KBSs, some recent research has been undertaken in integrating KE technology into general-purpose Software Engineering (SE) methods and tools [4, 7]. It is based on the observation that formalizing conceptual structures in ontologies is strongly related to modern object-oriented methodologies and languages such as UML [3] and its associated formal constraint language OCL [34]. Marrying SE and KE would allow to include

industrial tools, programming languages and modeling expertise into KBS development.

In this spirit, we propose a SE framework for the development of KBSs, the central idea of which is that abstract knowledge models can (and should) be embedded into the software architecture both on design and implementation levels. In our approach, ontologies are modeled using UML and automatically transformed into structure-preserving Java classes. Object-oriented reflection [10] is used to extract the high-level view of the ontology from the application at run-time. The resulting metadata is used to employ generic components for knowledge acquisition, consistency checking, schema translation and knowledge sharing. In support of our method, we have developed tools which have shown to be useful in a number of projects.

The following section discusses state-of-the-art KE approaches and their limitations. Section 3 gives an overview of our approach. Details on ontology design and implementation with UML and Java are provided in section 4. Section 5 presents tool support for knowledge acquisition and rapid prototyping. Our framework is applied to some case studies in section 6. Finally, we discuss benefits and shortcomings of our approach and make some conclusions.

2. Knowledge Engineering and Ontologies

First-generation expert systems, which tried to capture various types of knowledge uniformly in sets of rules, failed to produce consistent and reusable knowledge bases. Similar to research in SE, which has turned the art of ad hoc programming approaches into systematic design processes, the field of KE aims at supporting KBS development with clean-room modeling techniques and well-defined formal knowledge representation languages. This section will discuss modern KE approaches and their underlying concepts.

2.1. Ontologies and Problem-Solving Methods

Central notions of all KE methodologies are ontologies and problem-solving methods (PSMs). In the context of this paper, we define an *ontology* [33] as a shared specification of the objects and concepts that are assumed to exist in a domain of interest and the relationships and semantic constraints between them. The goals of building ontologies are to make knowledge explicit and reusable and to facilitate the communication between experts and the knowledge engineers during KBS development. PSMs capture knowledge about inference and reasoning behavior and enable the reuse and better understanding of strategic knowledge for different applications [28]. PSMs use task and method ontologies to specify the concepts and data types involved. PSMs can be used for a given domain by mapping the domain ontology onto the method ontology [23].

Most of the ontology representation languages that have been proposed in the literature, are based on either KL-ONE [5] or KIF [11]. The very popular frame-based [15] ontology representation scheme contains the following modeling primitives (cf. [21]):

- *Classes* represent the concepts, arranged in an inheritance hierarchy.

- *Slots* represent the attributes of the classes. Possible slot types are primitive types (integer, string, boolean), references to other objects (modeling relationships) and sets of values of these types.

- *Facets* contain meta-information on slots, such as comments, constraints and default values. Constraints are especially important, as they indicate semantic relationships between the ontology concepts.

- *Instances* represent specific entities from the KB.

Ontology languages allow to represent classes of concepts and their relationships in terms of expressions in formal logic. Once providing formal semantics this way, they support certain types of inference, such as subsumption, coherence, identity, compatibility and common specialization, which can help knowledge engineers in the construction of consistent knowledge bases (cf. [7]).

2.2. Knowledge Engineering methodologies

One of the most influential KE methodologies is CommonKADS [27]. Its main contribution is the architecture of the *Model of Expertise*, which represents knowledge on domain level, inference level and task level. The domain layer of this model is built around a domain ontology, whereas inference knowledge is captured in PSMs. The resulting models can be described using CML, a semi-formal, graphical notation, which is part of the CommonKADS framework. Although methods such as CommonKADS produced valuable insights on a theoretical level, their use in industrial projects is aggravated by the lack of tools and commercial support. As pointed out in [2], a major problem is missing support for the *transitions* between the abstract models and a high-quality implementation.

The Protégé [20] project aims at developing a tool set and methodology for the construction of domain specific knowledge-acquisition tools and KBSs from reusable components. In Protégé, ontology concepts are represented in a frame-based fashion by means of classes with slots of types such as `string`, `int` and `boolean` to store attribute values and relationships. The specific entities from the knowledge bases are stored as instances of these classes. The most recent version of Protégé [12], written in Java, provides a

class editor, a layout editor for configuring the user interface for knowledge acquisition, and a layout interpreter to edit and visualize the instances of the knowledge bases.

Since both KE approaches introduce their own formats and languages, their use in conjunction with conventional software architectures requires a considerable development overhead.

First, the responsible engineers have to get acquainted to these languages, which have their roots in AI and not in SE. Especially, the languages require a strong background in formal specification and are therefore unsuitable for knowledge modeling by domain experts without comfortable tools. Apart from that, it is questionable whether the full scope of inference capabilities provided by those languages is necessarily needed [7].

Second, interactions between the resulting knowledge modules and the remaining components, such as the user interface and inference algorithms, are difficult to implement. Either the knowledge models have to be re-implemented in the target programming language, or an additional layer of programming constructs has to be used to mediate between the different parts, like in Protégé. We used Protégé during the development of a Java-based KBS project which analyzes patient data in real-time to detect critical incidents in anesthesia [18]. The system mainly consists of a medical domain ontology and a generic PSM for process monitoring, which has its own PSM ontology. We found Protégé useful for conceptual modeling and knowledge acquisition, but had severe problems when we tried to link the domain ontology with other modules, such as those responsible for collecting and displaying patient data. Especially, we were missing support for linking the ontology concepts to the PSM algorithms. We argue that this loose coupling between knowledge models and the overall software architecture and process is a major weakness of current KE approaches.

To re-iterate, the KE community has provided valuable contributions on conceptual knowledge modeling, but the applicability of the current KE methodologies is aggravated by its weak amalgamation with industrial SE technology.

3. A Software Engineering process for KBSs

Our approach for developing KBSs is based on the object-oriented SE paradigm, which is supported by industrial standard methodologies, languages and tools. Our intention is to show that this technology provides a suitable platform for integrating KE concepts (especially ontologies). Since the basic object-oriented concepts are intended for any type of system, we propose appropriate extensions for KBS development. An overview of the steps involved in the development process and some of the main modeling artifacts is presented in figure 1.
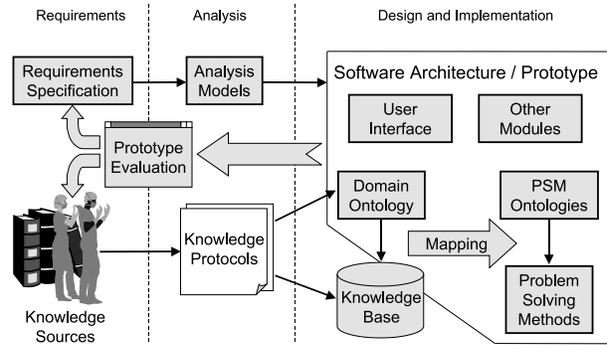
Figure 1. Overview of the development process for knowledge-based systems presented in this paper. Starting with requirements and informal knowledge sources, domain experts and engineers collaborate in the construction of shared domain ontology classes, which connect an abstract knowledge model to the remaining software architecture.

As pointed out in the introduction, KBS development requires the close collaboration between domain experts and software engineers. The upper part of figure 1 shows the process phases performed by the engineers, i.e. requirements analysis, design and implementation, followed by prototype evaluation in a feedback loop. We will not go into details on these phases, as they correspond to the processes propagated by standard methods such as Fusion [6] and the Rational Unified Process [13].

The steps performed by the domain experts are indicated in the lower part of the figure. Their tasks are to define system requirements, to provide and formalize knowledge, and to test the resulting prototypes. Requirements analysis includes the collection of available knowledge sources, such as text books and expert interviews. The knowledge available from these repositories can be condensed into informal knowledge protocols, such as annotated hypertext documents, which record factual knowledge as well as design decisions. KE models and methods, like structured interviews, can be employed for their development.

Based on this informal and weakly structured knowledge representation, the experts' task in the design phase is to construct a domain ontology and an associated knowledge base. Since domain experts usually have a rather informal domain view and little experience in conceptual modeling, they are supported by a knowledge engineer in this activity. Various methodologies for domain ontology construction have been proposed [19], but related approaches from SE and Entity-Relationship-Modeling are also applicable.

As a model representation and exchange language, the Unified Modeling Language (UML) [3] has been accepted as the standard formalism for object-oriented design and is widely supported by industrial development tools. The following section will provide details on the application of UML for ontology modeling and will show that the resulting ontology concepts can be automatically transformed into structure-preserving classes in an object-oriented language such as Java. These ontology classes, together with the remaining classes for user interface, PSMs and other functions, constitute the resulting KBS prototype. The knowledge-base of a prototype consists of (Java-) instances, which can be visualized and edited by domain experts with a knowledge-acquisition tool, as described in section 5. This tool is used to adapt the system to changes inspired by prototype evaluation.

To sum up, the focal point of our method is the object-oriented ontology model, which is used and edited by both engineers and domain experts and which merges knowledge-based concepts with executable software. In the following, we will focus on the development of ontologies. KE methods such as CommonKADS can be used for analysis activities and also for PSM design, which is not covered in this document.

4. Round-trip engineering of ontologies

We argue that in order to embed ontologies into the software architecture, the classes representing the ontology should be defined on the same meta-level as the remaining classes for user interface and so on. Otherwise, additional programming constructs are required to build up an intermediate layer between ontology and the other modules. However, since ad hoc approaches for designing ontologies as object-oriented classes lead to an inseparable mixture of knowledge-models and implementation details, a structure-preserving and reversible transition from ontologies to class models is required.

Round-trip engineering (RTE) [9] enables the automated translation of design models into an implementation language and back. Thus, changes on the model are instantly reflected in the source code, and the abstract models are updated after code modifications. RTE technology eliminates the need for manual transformations, and thus supports rapid prototyping and improves traceability of requirements. RTE is supported by modern UML-based CASE tools, such as Together/J [32]. In this section, we apply this technology to ontology development and integration into the KBS architecture.
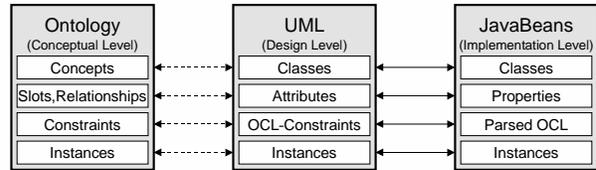

Figure 2. The conceptual view of an ontology and its representation with UML class models and in an object-oriented language such as Java. Since the formats share a common set of modeling primitives, the ontology's structure is preserved "round-trip" throughout the development life-cycle.

4.1. Ontology design with UML

UML provides graphical notations for various diagram types, such as use cases, interaction diagrams and class diagrams. Class diagrams mainly consist of boxes, displaying classes, their attributes and their methods. Connections between the boxes indicate relationships such as inheritance and association. In UML diagrams, the latter type of relationship can be annotated with multiplicity indicators, which constrain the minimum and maximum numbers of related entities. UML allows to attach annotations to diagram elements. These annotations may be comments on design decisions, but can also indicate constraints, which can be informal text or expressions written in the Object Constraint Language OCL [34]. The intention of this language is to facilitate the specification of model properties in a formal, yet comprehensible way. OCL expressions can also be used to express the constraints imposed by the multiplicities of relationships. UML diagrams can be stored and distributed in the XMI [22] format.

The left half of figure 2 shows the reflection of an abstract ontology view in a UML class model.

- *Concepts* from the ontology are represented as UML classes. These classes can be extended by additional members, which are not part of the ontology view. For example, domain ontology concepts can be enriched by methods for graphical display and PSM ontology classes can be directly related to the algorithms implementing the reasoning behavior.

- *Slots* (and if applicable their default values) are modeled as attributes of these classes. Slots that represent relationships should be modeled by corresponding UML relationships. In order to support the re-extraction of the abstract ontology view from such classes, the attributes representing ontology slots are
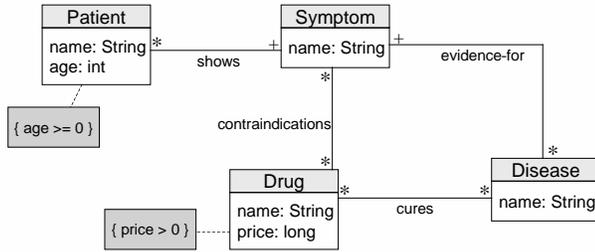
Figure 3. A small example ontology in UML. The expressions in curly brackets are OCL constraints associated to the classes.

annotated in a predefined way (in our case as "properties" – details on this can be found in subsection 4.3). This allows to distinguish between slots and implementation details. It is possible to extract the set of ontology concepts from the whole class structure, by starting with a known "root" ontology concept and traversing the slot-representing attributes recursively.

- *Constraints* are attached as OCL expressions to the corresponding attributes and classes.

- *Instances* from the knowledge base could be represented in UML class diagrams. However, current CASE tools provide little support for instances and the diagrams are prone to becoming unintelligible as the number of objects grows. A special-purpose knowledge acquisition tool, like the one presented later, is more suitable here. However, the option of putting instances into UML diagrams and the resulting XMI files simplifies knowledge base storage and exchange.

Figure 3 shows an example ontology on medical concepts in UML notation.

4.2. Ontology implementation from UML models

In order to enable the smooth transition between the ontology design in UML and the underlying programming language, we now define a mapping between UML class models and object-oriented concepts, as indicated in the right part of figure 2. In general, UML has some shortcomings coping with RTE [9], because some details on the dynamic behavior of code can not be adequately represented in UML. However, these limitations do not concern the static ontology models. It is straight forward to translate UML classes, attributes, relationships and instances to according object-oriented concepts. This feature, as well as the reverse engineering of code into UML diagrams is an integral part of standard CASE tools.

More difficult is the task of converting constraints and the multiplicities of relationships, as the UML does not provide guidelines for their implementation. Since multiplicities can be expressed in terms of OCL expressions, we only need to consider the transitions between OCL and an object-oriented language. As OCL expressions are not part of the programming languages, they either have to be compiled to source code, or interpreted at run-time. OCL compilation can lead to better run-time performance compared to interpretation, but limits the evaluation of constraints for high-level operations and is not supported by current CASE tools. Interpretation requires the programming language to support generic access to class attributes – a feature provided by reflective languages such as Java (cf. the following subsection). As we are using Java in our approach, we do not directly represent the OCL constraints and multiplicities in the target programming language, but employ an OCL interpreter that is initialized with the original expressions from the model. The constraints are imported from the XMI files and parsed into OCL expression trees when the system starts. These trees are then available for various types of inference on ontologies. Thus, constraints support knowledge acquisition by preventing the definition of inconsistent knowledge bases. Section 5 provides details on this.

Because structural operations such as ontology constraint checking and knowledge acquisition need access to metadata on the ontology, the implementing language has to enable access to such meta-information at run-time. The following section will show that object-oriented *reflection* [10] is the key technology for that purpose.

4.3. Ontology implementation with JavaBeans

For the implementation of ontologies, we chose Java, although many of the concepts being presented are also applicable for similar platforms. A main benefit of Java, compared to other main-stream programming languages such as C++, is its good support for reflection. Reflective programs can reason about their own structure at run-time. They are able to analyze the available classes, their attributes and their methods and use this structural metadata to access and invoke class members on a generic level. In the context of our KBS development framework, reflective features are used to implement generic functionality on ontologies, such as knowledge visualization, consistency checking, schema translation and knowledge sharing.

Java includes classes such as `Class`, `Method` and `Field`, which can be used to retrieve information on the structure of run-time objects. The *JavaBeans* API [29], which is backed by this reflection mechanism, was initially developed to support the development of sharable components with well-defined interfaces. The JavaBeans designers mainly aimed at visual programming with components

such as buttons and dialogs, which can be easily configured by builder tools. The API specifies how a JavaBeans class exposes its features, so that tools can analyze and present them. For each field, which is to be published (called a *property*), only suitable `get` and `set` methods must be written. JavaBeans properties can be of any primitive type, such as `int`, `double` or `boolean`, or references to objects. Properties are either *single* or *indexed*. Indexed properties represent arrays of the above types. Properties are said to be *bound*, if they deliver a `Property-ChangeEvent` after their value has changed. Bound properties are simply implemented by adding an event dispatching call to the end of the `set` method.

Another feature of JavaBeans classes is support for visual editing tools. For each property of a class, it is possible to specify a `PropertyEditor` component, which graphically presents the property value in a human readable form. Thus, for example, sliders can be used to enter per cent values. Java uses reflection to extract metadata, such as the available properties and their types. Thus it allows to write code for property access when only the name of the property is given as a String.

Due to the obvious similarities between JavaBeans properties and ontology slots, the JavaBeans API lays the foundation for mapping ontologies to the implementation.

- *Concepts* and the corresponding UML classes are represented by one JavaBeans class each.

- *Slots* / attributes are stored as JavaBeans properties. In support of ontology visualization and editing, we declare all properties to be bound, so that events are dispatched on modifications. Using JavaBeans properties instead of simple attributes for the implementation allows to distinguish between attributes implementing ontology slots and those representing implementation aspects. The CASE and development tools we tested provided specialized support for editing JavaBeans properties comfortably.

- *Constraints* are not implemented as Java code, but parsed to OCL expression trees that use reflection to access the slot values of the specified objects. As long as RTE of OCL expressions is not supported by CASE tools, own conventions have to be introduced to make constraints available to Java objects. The easiest way is to leave them in the XMI files, so that the UML diagrams are the only artifacts where constraints have to be edited.

- *Instances* are mapped to JavaBeans objects.

4.4. Ontology exchange and mapping

Ontologies represented in UML and Java are sharable on various levels. First, the ontology implementing Java classes can be reused, although some implementation aspects might have to be removed, as they are custom-tailored for the specific application. Therefore, exchange of the ontology on UML level is much more promising. Sharing of UML models also allows to map an ontology to different languages and implementations, so that communicating agents can have the same ontology interface, but different local implementations. Due to the common core of frame-based languages, it is also possible to define mappings between other popular ontology representation languages and thus to include existing KE technology into the process.

The knowledge-bases, consisting of a network of instances, can be stored either in XMI, in Java's own Java-Beans archive format [31] based on XML or using a mapping between JavaBeans and relational databases, such as in the JavaBlend technology [30]. Since XML is being discussed as a promising candidate for describing data and metadata of information repositories such as the internet, our approach is therefore compatible to the most important industrial standards.

XML also provides the schema translation language XSL-T [35], which can be used to describe structural mappings between XML documents. This language could be used to implement ontology sharing and translation by applying schema transformation rules defined in XSL-T to ontology instances that were temporarily converted to XML documents. Thus, for example, domain ontologies can be transformed to method ontologies, and knowledge can be communicated in a multi-agent setting. An alternative approach to mapping ontologies based on generic mapping classes and reflection has been described by the authors in a previous paper [17].

5. Tool support

Ontology design and the design of the remaining project classes can be performed using off-the-shelf CASE tools. A problem is that CASE tools – and UML – provide too many features, that are not needed for ontology design. Thus, these tools are best applied if knowledge engineers and domain experts make ontology changes together. The *relative interaction hypothesis* [26] states that some kind of dependency exists between the structure of domain knowledge and the types of tasks it is applied to. Therefore, ontology design is anyway typically accomplished in a combination of the domain and engineering points of view and changes in the ontology require changes in algorithms, mapping rules or other modules from the system.
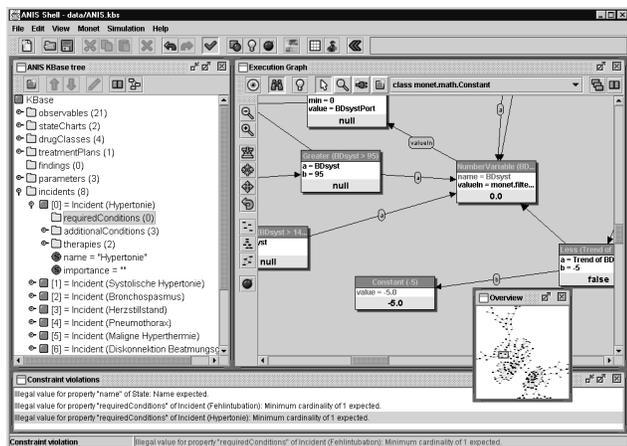
Figure 4. A knowledge acquisition tool for an anesthesia information system, based on a generic JavaBeans editor. The knowledge base is displayed as a tree (left part). The currently violated constraints are shown in the bottom window. The graph shows instances from a PSM for process monitoring.

In contrast to structural ontology changes, editing and maintaining the knowledge base instances can usually be performed by domain experts alone. We have developed an extensible knowledge acquisition shell for Java-based projects (figure 4). The shell can be applied to *any* Java-Beans data structure, because it uses reflection to retrieve metadata on the available classes. It can be used to visualize and edit the instances from the knowledge base as a tree, as a graph, or by means of fill-in forms. After each modification, the constraints are tested and violations are displayed in an extra window. A mouse click on the constraint moves the cursor to the respective instance or slot.

Due to its open architecture, the shell provides a flexible foundation for special-purpose knowledge acquisition tools. If a special property editor is defined for a JavaBeans class, it is automatically used instead of the generic component. Custom windows, for example to display internal PSM reasoning processes, can also be included. Because the run-time Java objects representing the ontology are identical to the ontology concepts at design-time, it is possible to invoke the knowledge editor while the KBS is running.

In support of a smooth transition between the abstract knowledge models and the instances from the knowledge base, the tool has features to link parts of the knowledge base with knowledge protocols and requirements specifications, such as the hypertext documents written during the analysis phase of the development process. Thus it is easier to trace or update requirements and design decisions.

6. Case studies

We have based various academic and industrial KBS projects on our methodology. One project [18] aimed at the development of a clinical information system, which analyzes incoming patient data (e.g. blood pressure) to detect critical incidents during anesthesia. A knowledge engineer and some domain experts collaborated in the specification of a medical domain ontology, which had to be redesigned various times until a suitable KB could be entered. For that purpose, we extended the generic knowledge acquisition shell described above by components that simulate patient data. Next, we developed a generic PSM for monitoring data streams. In order to map the domain ontology to the PSM ontology, we added a map method to each ontology class, which delivers PSM ontology instances for domain instances (In this project, there was no XSL-T support for Java available yet). The graph view in figure 4 shows some of the instances used by the PSM. In order to visualize the internal processes and computational states of this PSM, we added special windows to the knowledge editor, displaying the streams of patient data and their impact on the inference mechanisms. Parallel to the construction of the KB, software developers implemented the graphical user interface and other modules that access the ontology classes, for example to feed them with patient data. Therefore, both ontology designers and software engineers modified the same classes, although on different levels of abstraction.

In another case study, we use our method to develop a knowledge-based process construction kit for the capture and dissemination of process know-how [25]. The requirements for this system are drawn from the domains of automotive engineering and plant construction. One task of the system is to derive generic process design rules by evaluating typical construction sequences in the context of a given process. Since the task of observing a process designer during his construction task is related to monitoring a patient in a clinical setting, we were able to re-use many concepts from the ontology of the anesthesia information system. Next, we extended the generic graph editor from our workbench, and managed to provide a prototypical process editor within a few hours of programming.

7. Discussion

The goal of this paper was to show that combining KE and SE methodologies is feasible and of great promise. We will now discuss some of the expected benefits, assumptions and limitations of our approach.

The method for KBS development presented in this paper is very application-oriented by relying on standard tools, languages and guidelines. As standards such as UML, XML and Java are rapidly gaining importance both in industrial

and academic projects, so increases the potential of our framework. As a good example for tool interoperability, our approach is expected to work very well with the commercial multi-agent construction tool AgentBuilder [24], the developers of which (independently) also decided to implement the agent ontologies as JavaBeans.

In the presented framework, ontologies are much closer embedded into the software architecture than in other KE approaches. This allows programmers to add fields and methods for interactions between knowledge-based and other modules. Also, the overhead of implementing intermediate ontology access layers (such as Active Object Models) is reduced. De Michelis et al. [8] argue that "reflective systems have the potential to eliminate the need for meta languages and meta models to manage links between the system and its models at higher levels of abstraction". By eliminating the need for intermediate layers by means of reflection, the resulting systems also have a better run-time performance. A limitation of our modeling approach is that it is restricted to features provided by the target programming language. For example it is impossible to redefine the meta-classes themselves, like it is possible in Protégé (a comparison of the latter approach with our reflective mechanism can be found in [16]).

Since UML models are automatically mapped to structure-preserving constructs of the target programming language, design and implementation artifacts are comparable. This allows very smooth transitions between the various models. The transitions between design and implementation model are fully reversible, by applying RTE technology. This technology rapidly produces prototypes which can help development teams to better understand the requirements and to test the knowledge bases. As requirements tracing from the initial knowledge models is a major concern in KBS development, our approach reduces the number of errors resulting from manual transitions. With RTE, developers can adapt the system quickly either on model or code level, with no need to run through various design models. Thus, the approach supports both cleanroom SE methods and light-weight methodologies such as Extreme Programming [1], the paradigm of which is that source code should be produced early and refined later, when the requirements have become more stable.

Finally, we have shown that the reflection-based implementation technique enables the reuse of generic components, such as JavaBeans editors, constraint checkers and schema translators.

According to our experience in case studies, these benefits lead to considerable cost and time savings. The case studies suggest that the framework is functional for small to medium-sized projects. However, more feedback from independent sources is required to fully evaluate the concepts. Since SE technology is well-prepared for large projects, we expect the approach to be scalable for general industrial use.

Our approach is limited by a two assumptions, that may reduce its applicability in some projects. First, the framework is optimized for object-oriented languages only, especially those with reflective features. Second, we assume that the *relative interaction hypothesis* [26] (as discussed in section 5) holds, so that changes of the ontology structure are best made in a collaboration between domain experts and (knowledge) engineers. One reason for this is that current CASE tools are very general and not optimized for knowledge modeling, so that inexperienced domain experts would be confused by the amount of available features.

8. Conclusion and future work

In this paper, we have presented a SE framework for KBSs. By embedding ontology construction into an object-oriented development process, abstract knowledge models are smoothly integrated into an executable system. Our framework is based on RTE, maintaining meta-information on the ontology between design and implementation. Reflection is used to extract this meta-information at run-time so that generic components for various tasks on ontologies can be reused. We have provided evidence that UML is indeed a "universal" modeling language, in so far that its modeling primitives are suitable for knowledge representation. The main difference between standard object-oriented SE methodologies and the KBSs development process is the addition of explicit knowledge acquisition activities, which result in a network of objects or instances. The fundamental difference to standard KE technology is that we do not only regard KBS development as a knowledge modeling activity, but as a software construction process, with knowledge models as one type of components among others. The excellent support for prototyping and the use of reflection can be regarded as the main innovations of our approach.

A dedicated strength of our approach is the integration of common standards, instead of designing proprietary representations. This brings us in the position to reuse a broad variety of SE techniques and in particular user interfaces to such standards. Yet, additional efforts are needed to develop custom-tailored tools and more intuitive, easy-to-use interfaces for use by domain experts. One goal of our future work is to provide a custom-tailored alternative to standard UML CASE tools for ontology modeling. We are currently building such an ontology editor based on our JavaBeans editor. This tool allows to edit instances of a meta-ontology, containing concepts such as classes, slots and constraints. With this tool, domain experts can easily define their ontologies and export them to UML in XMI files, which are then used by CASE tools to generate and update the ontology Java classes automatically.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] R. Benjamins, D. Fensel, C. Pierret-Golbreich, E. Motta, R. Studer, B. Wielinga, and M. Rousset. Making Knowledge Engineering Technology Work. In *Proc. of the 9th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE-97)*, Madrid, Spain, 1997.

[3] G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] P. Borst, E. van der Wal, and P.-H. Speel. Developing a Library of Domain Models Using Commercial Tools. In *Proc. of the Knowledge Acquisition Workshop (KAW-99)*, Banff, Canada, 1999.

[5] R. Brachman and J. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.

[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[7] S. Cranefield and M. Purvis. UML as an Ontology Modeling Language. In *Proc. of the IJCAI-99 Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

[8] G. de Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, J. Schmidt, C. Woo, and E. Yu. A Three-Faceted View of Information Systems. *Communications of the ACM*, 41(12):64–70, 1998.

[9] S. Demeyer, S. Ducasse, and S. Tichelaar. Why Unified is not Universal: UML Shortcomings for Coping with Roundtrip Engineering. In B. Rumpe, editor, *Proc. of the Second Int. Conf. on The Unified Modeling Language (UML'99)*, Fort Collins, CO, 1999.

[10] J. Ferber. Computational Reflection in Class-Based Object-Oriented Languages. In *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA-89)*, New Orleans, LA, 1989.

[11] M. Genesereth and R. Fikes. Knowledge Interchange Format, Version 3.0, Reference Manual. Technical Report, Logic-92-1, Computer Science Department, Stanford University, 1992.

[12] W. Grosso, H. Eriksson, R. Fergerson, J. Gennari, S. Tu, and M. Musen. Knowledge Modeling at the Millennium (The Design and Evolution of Protégé-2000). In *Proc. of the Knowledge Acquisition Workshop (KAW-99)*, Banff, Canada, 1999.

[13] I. Jacobsen, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[14] M. Jarke. Requirements Tracing. *Communications of the ACM*, 41(12):32–36, 1998.

[15] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42, 1995.

[16] H. Knublauch. Three Patterns for the Implementation of Ontologies in Java. In *Proc. of the OOPSLA-99 Metadata and Active Object-Model Pattern Mining Workshop*, Denver, CO, 1999.

[17] H. Knublauch and T. Rose. Reflection-enabled Rapid Prototyping of Knowledge-based Systems. In *Proc. of the OOPSLA-99 Workshop on Object-Oriented Reflection and Software Engineering*, Denver, CO, 1999.

[18] H. Knublauch, M. Sedlmayr, and T. Rose. Knowledge-Based Decision Support in an Anaesthesia Information System. In *Proc. of the ESCTAIC Annual Meeting*, Glasgow, UK, 1999.

[19] F. López and A. G. Pérez. Overview of Methodologies for Building Ontologies. In *Proc. of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods*, Stockholm, Sweden, 1999.

[20] M. Musen, S. Tu, H. Eriksson, J. Gennari, and A. Puerta. PROTÉGÉ-II: An Environment for Reusable Problem-Solving Methods and Domain Ontologies. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-93)*, Chambery, France, 1993.

[21] N. Noy and M. Musen. An Algorithm for Merging and Aligning Ontologies. In *Proc. of the AAAI-99 Workshop on Ontology Management*, Orlando, FL, 1999.

[22] OMG. XML Metadata Interchange (XMI). `http://www.omg.org/cgi-bin/doc?ad/98-10-05`, 1998.

[23] J. Park, J. Gennari, and M. Musen. Mappings for Reuse in Knowledge-Based Systems. In *Proc. of the Knowledge Acquisition Workshop (KAW-98)*, Banff, Canada, 1998.

[24] Reticular Systems. AgentBuilder 1.2. `http://www.agentbuilder.com`, 1999.

[25] C. Rupprecht, M. Fünffinger, H. Knublauch, and T. Rose. Capture and Dissemination of Experience about the Construction of Engineering Processes. In *Proc. of the 12th Conference on Advanced Information Systems Engineering (CAISE)*, Stockholm, Sweden, 2000.

[26] A. Schreiber, B. Wielinga, R. de Hoog, H. Akkermans, and W. van de Velde. CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, 1994.

[27] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. van de Velde, and B. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT-Press, 1999.

[28] R. Studer, D. Fensel, S. Decker, and V. Benjamins. Knowledge Engineering: Survey and Future Directions. In *Proc. of the 5th German Conf. on Knowledge-based Systems*, Würzburg, Germany, 1999.

[29] Sun Microsystems. JavaBeans Specification. `http://java.sun.com/beans`, 1997.

[30] Sun Microsystems. JavaBlend. `http://www.sun.com/software/javablend`, 1999.

[31] Sun Microsystems. Long-Term Persistence of JavaBeans using XML. Currently under review, 2000.

[32] TogetherSoft. Together/J. `http://www.togethersoft.com`, 2000.

[33] M. Uschold and M. Gruninger. Ontologies: Principles, Method and Applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.

[34] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[35] World Wide Web Consortium. Extensible Stylesheet Language (XSL). `http://www.w3.org/Style/XSL`, 1999.