OBSE – an approach to Ontology-based Software Engineering in the practice

Andrej Bachmann, Wolfgang Hesse, Aaron Russ (University Marburg)
Christian Kop, Heinrich C. Mayr, Jürgen Vöhringer (University Klagenfurt)

Philipps-University Marburg
Hans-Meerwein-Strasse
35032 Marburg
{rodionov,hesse,russa}@mathematik.uni-marburg.de

University of Klagenfurt
Universitätsstraße 65 - 67
9020 Klagenfurt
{chris,heinrich,juergen}@ifit.uni-klu.ac.at

Abstract: In this article we present a new approach to Ontology-based Software Engineering (OBSE) meant for practical use in enterprises and industrial projects. Following this approach, Software projects are no longer driven only by requirements and models but also by one or several ontology/ies covering their application domain. Our main thesis says that OBSE can offer similar opportunities and benefits for re-engineering and re-use in the early phases of software development as object orientation does for the later ones. OBSE is to be supported by tools which integrate ontologies in the SE process. A prototype of such a tool – based on the KCPM and EOS methodologies – is presently being developed in a joint project of our groups.

1 Introduction

Ontologies have been introduced as a key concept in informatics in the last decade of the previous century when the rapid growth of the internet and its services created new demands on automated agents and similar devices which require facilitated and encompassing access to domain knowledge of various application domains. Gruber has defined *ontology* as an *explicit specification of a conceptualization* [Gru 95]. Applications of ontologies in Informatics comprise the fields of Artificial Intelligence, Agent systems, Database & Information systems, Web Technology and other fields.

In the *Software Engineering (SE)* field conceptualisation has played a major role for long time, e.g. during the *analysis, modelling and design* phases of software development, where all relevant entities of the application domain with their features and relationships have to be conceptualized. This way, considerable portions of domain knowledge are elaborated in almost every application-oriented software project. Similarly, large portions of technical and implementation-oriented knowledge are worked out during the *detailed design, implementation, test & integration* phases. However, whereas *object orientation (OO)* technology has led to major achievements in re-using the latter kind of knowledge this is still a desire and challenge as far as the domain knowledge relevant for the early phases is concerned.

Ontologies seem to be an appropriate concept for describing portions of domain knowledge which is to be re-used across several software projects. Thus we aim for a new approach for integrating ontologies in the SE process. Following this *Ontology-*

based Software Engineering (OBSE) approach, software projects are no longer driven only by requirements and models but also by one or several ontology/ies covering their application domain (cf. fig 1).

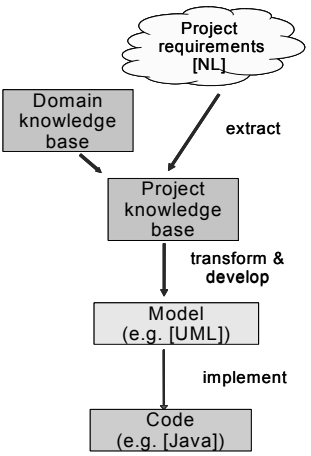

Fig. 1: An ontology-based software processing

Our main hypothesis is that by combination of Knowledge and Software Engineering techniques OBSE can offer similar opportunities and benefits for re-engineering and re-use in the early phases of software development as object orientation does for the later ones. Of course, such an approach has to be supported by tools for software engineers who want to integrate ontologies in their SE process.

Two major questions arise when we investigate approaches to OBSE in more detail:

(1) What is the appropriate form (and language) for expressing ontologies in the SE context? **and**

(2) How can the SE process be extended to an OBSE process including the use and evolution of ontologies?

For dealing with both questions, previous work of the two groups authoring this article can usefully be employed:

- *The Klagenfurt Conceptual Predesign (KCP)* method offers *glossaries* as an appropriate and useful instrument to deal with ontologies in the early phases of software development.
- The (Marburg-based) method for *Evolutionary, Object oriented Software Development (EOS)* offers a Software Process model which is flexible and wide enough to cover ontology-based processes as well and allows for a unique treatment of software and ontology engineering processes.

In our view, the OBSE process should be a combination of both (Software and Ontology Engineering) life cycles following some sort of *rendezvous* principle: Software Engineering projects inherit from existing ontologies in the early (analysis and modelling) phases and offer (parts of) their results for further ontology development and evolution. The first step allows re-use of domain knowledge whereas the later promotes re-use of project specific knowledge. The EOS model serves as a joint framework for defining and supporting the OBSE approach.

Furthermore we argue that the appropriate form for including domain knowledge from ontologies in the SE process is the *glossary form* making the KCP method and tools a key basic technology for OBSE. Since glossaries are modular, flexible and easy to handle for users, domain experts and engineers, they seem to be a most appropriate form for documenting reusable knowledge.

In the subsequent sections, we will first briefly compare Software and Ontology Engineering processes and then define our OBSE process model based on the KCP and

EOS approaches. In the last sections we will deal with tool support for OBSE and include a rough sketch of the tool prototype under construction.

2 Software and Ontology Engineering process: brief comparison

Ontological Engineering has been advocated by Mizoguchi [Miz 98] and analogies (as well as differences) with Software Engineering and their processes have already been discussed, e.g. in [G-L 02] and [Hes 05]. In the following table, both fields are compared and of some of their outstanding characteristics and properties are listed (cf. Fig. 2).

	Software Engineering	**Ontology Engineering**
Target groups	• Software managers, engineers and users engaged in a particular project	• Domain experts, ontology builders and users dealing with many projects in a particular domain
Principal requi-rements	• Functional requirements regarding system procedures, correct output etc. • Quality requirements re. user friendliness, reliability, performance ..	• Trustability and consistency • Compatibility and accessibility from many projects and applications
Project duration	• determined, limited for one project	• undetermined, unlimited
Process structure	• often sequential: phases, activities, but also iterations and cycles • Sub-processes for components or increments	• mostly cyclic, cycles maybe grouped in phases • Sub-processes for developing or revising subdomains
Process models	• Waterfall, incremental, component-based, prototyping, spiral-like	• incremental, component-based, evolutionary
Concepts, languages and tools	• Use cases, natural language, modelling & programming languages (e.g. UML), diagrams, pseudo code • Tools: Editors, modelling tools, compilers	• Natural language, glossaries, tables, semantic networks, topic maps, conceptual graphs, ontology languages • Tools: Ontology editors, modelling tools
Results and products	• project-specific, (relatively) short-term oriented, usable for particular application	• spanning many projects, long-term oriented, re-usable, "sharable" among many organisations and projects

Fig. 2: Some characteristics of Software and Ontology Engineering

As the table shows, ontology development resembles software development in various respects but there are also significant discrepancies between the two kinds of processes, e.g. resulting from different target groups, contexts and requirements. For a more comprehensive discussion we refer to [Hes 05].

3 KCPM: A glossary-based approach to Software and Ontology Engineering

In order to motivate the *KCP method (KCPM)* as a missing link between software requirements and software modelling/design we will briefly present our glossary approach, the concepts and representation forms of KCPM. Afterwards the appropriateness of KCPM will be discussed.

KCPM was introduced to support the requirements elicitation process. As described in the previous section there are a lot of similarities between software engineering and ontology engineering. During the first phases of *requirements engineering* and *knowledge acquisition* the developers (ontology builders and software engineers, resp.) have to communicate with experts[1]. Performing this task the engineer is more like a doctor who has to ask the right questions or like a pilot who has to check that everything is working before he starts the engine. The paradigm of such a check list which supports the task to formulate the right questions directly leads to the idea of using glossaries as a concept for representing requirements. In the KCP methodology, a glossary is employed as the central knowledge base for gathering, storing and communicating domain knowledge during the requirements capture and modelling phases of software projects [M-K02].

In particular glossaries have the following advantages:

- Domain experts are mostly familiar with glossaries since they use them in their daily work.
- It is easier to find a gap in a glossary-like specification (the regarding column is empty). Thus a glossary is like a check list.
- Information that belongs together (e.g. regarding the same concept) is associated with that concept. Thus the information is collected in a very compact manner.
- The structure of a glossary is standardised and predefined.
- The semantics of the key terms of glossaries (e.g. the column names) are predefined.
- The glossary type (e.g. thing type glossary, operation type glossary) as well as the several columns within these glossary types provides a first classification of the collected information.

3.1 Small set of modelling concepts

The glossary is built up by few kinds of (table-like) type descriptions the most important of which are: *thing types* and *connection types*. In order to support the glossary building task, linguistic techniques such as natural language text analysis are employed and supported by corresponding tools [FKM+ 00]. Glossaries may be transformed into conceptual models or UML-like class structure diagrams according to a set of laws and transformation rules in a *semi-automated* way.

---

[1] We assume that every person is an expert on a certain domain. In the most specific way he/she is the expert of the tasks he/she has to do in an enterprise.

Thing-type is a generalisation of the UML concepts class and attribute. Thus, typical thing types are e.g. *author*, *book*, *contract* as well as descriptive characteristics like *customer name*, *product number*, *product description*. It seems to be easy to decide, which of the above examples is a class and which one is an attribute, but what about a concept used in a domain which is not well known by the designer (e.g. the concept *ICD10* in the medical domain). Following KCPM the question whether the concept is a class or an attribute is not a primary question but this will be decided later and be supported during the mapping process. Instead the system analyst can concentrate on gathering additional information for that concept, which is much more important during requirements analysis. Meta-attributes which head the glossary columns (e.g. *Examples, Synonyms, QuantityDescription*) give hints to ask the right questions.

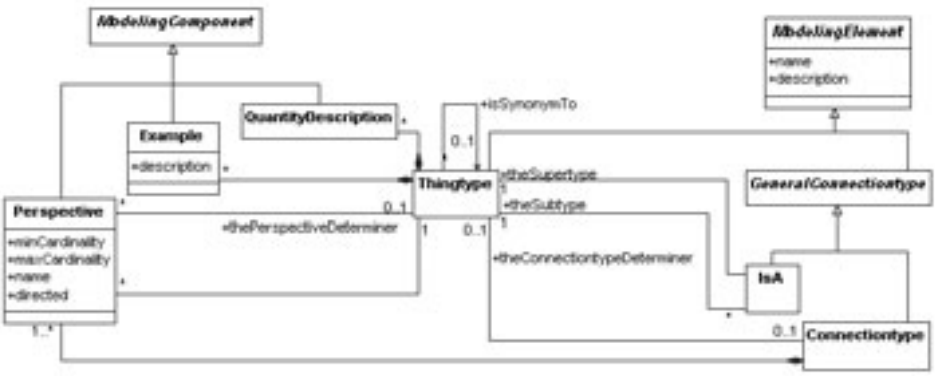

Fig. 3: Overview of the KCP meta model

Things are related within the real world. To capture this, the KCPM model introduces the concept of **connection-type**. Two or more thing-types can be involved in a connection-type. This is based on the NIAM (ORM) object/role model [N-H 89]. A sentence (business rule) leading to a connection type could be the following: *Authors* write *books*. The model is open for specific semantic connection-types (possession, composition, generalization, identification etc.) e.g. *An ISBN number* identifies *a book*.

This glossary approach works similar for all the other KCPM concepts (connection type, operation type etc.) which are described in other papers. In the meta schema concepts and columns (meta attributes) are distinguished in the following way. KCPM concepts are derived from the class *ModelingElement* and columns from *ModelingComponent*.

3.2 KCPM as a link between domain ontologies and SE

As was mentioned before, KCPM was introduced as a requirements modelling language. However looking at the modelling concepts and the representation concepts of KCPM, KCPM can be also seen as a link between Ontologies and Software engineering. In order to motivate this assumption it has to be shown that

(1) There is a general relationship between KCPM and ontologies

(2) KCPM can be used for conceptual models in the software engineering domain.

To justify the first statement we need to answer first the following questions: What is the purpose of an ontology? What are possible ontology representations?

Since Gruber's article [Gru 95] ontology is understood as a means for knowledge sharing. In [Gua 98] domain and task ontologies describe the vocabulary related to a generic domain. This distinguishes domain ontology from a conceptual model where the vocabulary has to be refined for the project specific purpose.

For the second question we have to take a look at the several representations of ontologies. These representations range from lexicons, notion lists, and topological maps to formal specifications. Most often glossaries are used to describe the notions.

Comparing this with the KCPM approach we can conclude: KCPM has a representation concept that fits very well into possible representation concepts of ontologies. Furthermore, in the previous section the advantages of such a representation were already stated. These advantages can also be applied to ontologies.

As static concepts KCPM offers just thing types and connection types. From our experience in many modelling projects, we learned that the distinction between classes and attributes is often artificial, premature or project dependent. A notion which might be modelled as a class in one project (e.g. *address*, *driver*) might be modelled also as an attribute in another project. If we abstract from this distinction using thing types for both in common, then this fits much better to the ontology level where the involved parties have to concentrate on getting a *shared knowledge and understanding* of a domain.

To motivate the second statement from above, we refer the reader to [M-K 02] where we have described in detail how thing types and connection types can be mapped to conceptual models used in the SE domain (i.e. UML diagrams). Furthermore it was explicated in [V-M 05] that integration on the conceptual level (e.g. using KCPM) has advantages compared to integration in the later phases of software development. This has also a beneficial impact on the process of shared knowledge generation. If two or more involved parties want to adjust their knowledge to a common shared knowledge it is much better to abstract from terms like class and attributes.

3.4 General Overview of the OBSE cycle

Based on these connections between KCP glossaries, UML, and ontologies, a unification and combination of the ontology development life cycle for a certain domain on the one hand side and software project life cycles concerning the same domain on the other is promising. A first glance on the combined life cycles is shown in fig. 4. The key for this unification is the use of KCP glossaries in both life cycles: On the ontology side, the domain knowledge is captured in a glossary before it is (partly automatically, see above) transformed to UML or some other ontology language (OL) representation. On the software project side, project specific domain knowledge is extracted from the requirements after elicitation and stored in a glossary-like knowledge base which is transformed to UML models in the described way and then further to code of some *programming language (PL)*.

In our unified process, KCP glossaries are used in order to support the requirements and ontology engineering processes as well as the exchange between both of them. Since in this phase requirements are mainly collected through user interviews and the study of *natural language (NL)* documents, ambiguities can easily occur and lead to communication problems. On the other hand, domain ontologies usually contain data that is reviewed by domain experts and that describe the important concepts and relationships of the specific domain.

These data can be used in various ways to replace or to complement information gathered in the "normal" requirements engineering process. For example, information that was collected through usual requirements engineering techniques might be verified while matching it against the ontology data. This way, discrepancies between the ontology description and the data from other sources might be identified. These conflicts are often caused by ambiguities or imprecision in the requirements documents and must be resolved. Moreover, the ontology data might complement the previously gathered requirements. This can be accomplished combining the domain glossary and the project glossary by *schema integration* (cf. chap. 3.3).

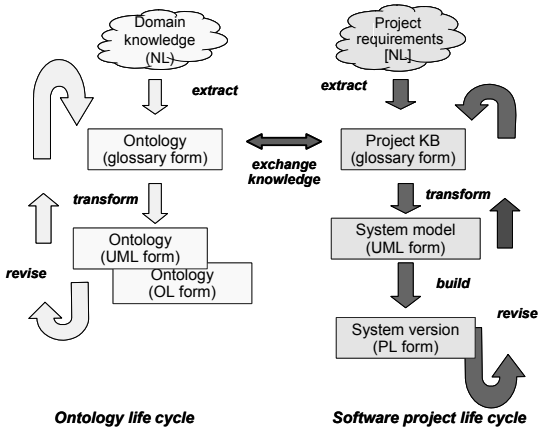

Fig. 4: Ontology and software project life cycles combined in a rendezvous manner

This combined and integrated ontology can be modified in order to meet project specific needs. Using this ontology the project runs through the following design and implementation phases of the software development life cycle. During the final project phases domain knowledge that has been gained during the project and which was not yet part of the domain ontology or which should be used for its revision can be exported and then used within an integration process modifying the original domain ontology.

4 The EOS model and its use for a combined OBSE process

4.1 Basic concepts of the EOS model

In order to obtain a uniform view on both Software and Ontology Engineering processes as sketched above, the EOS model can successfully be used (cf. [Hes 03], [Hes 05]). It has been developed in order to support *Evolutionary Object oriented Software Develop-*

ment and it offers a high degree of flexibility and scalability for both software managers and engineers when dealing with complex projects which include many components and highly concurrent development processes. Its use for Software Engineering projects has been described in detail elsewhere (cf. e.g. [Hes 96], [Hes 97], [Hes 03]). Among its key concepts are:

- *Component-based structure and architecture-driven, uniformly structured development cycles:* Unlike most of its traditional predecessors, the EOS model binds development cycles to the building blocks of the system architecture. All development cycles consist of the four main activities analysis, design, implementation and operational use – irrespective of its occurrence at the system, component or module/class layer (cf. right part of fig. 5). This way, development processes become highly scalable and flexible.
- *Multiple, mostly concurrent development cycles and evolutionary software development:* Re-development or revision cycles may be activated and performed on demand at any architectural level and thus system evolution is encouraged and supported by the EOS model. Concurrent development cycles are synchronised by means of revision points, i.a. predefined points in time where certain activities have to be finished and their results are available for review or delivery.

4.2 Ontology development described in EOS terms

Ontologies are preferably structured as hierarchies (cf. [Gua 98]), e.g. consisting of the three levels:

- universal ontology
- discipline ontologies
- domain ontologies

Further decomposition may lead to smaller units – let us call them *ontology components (OC's)* in analogy to the EOS terminology. Ontologies are in a continuous process of *evolution* – this leads to the requirement of independent, often concurrent OC development cycles. These can be well defined using the general EOS schema for describing development cycles:

- *Ontological analysis* aims at defining the OC and delimiting its boundaries, identifying potential applications, analysing the relevant terminology, building a taxonomy, describing terms as glossary entries, and dissolving terminological conflicts. The analysis results in a first version of the OC glossary.
- *Ontology design* deals with defining a (sub-) structure of the OC, defining facts and rules, building UML maps of the OC, check-ups and comparisons with other glossaries, modifying and (re-) structuring the taxonomy and particular glossary entries, dissolving terminological conflicts.
- *Ontology implementation and integration* aims at translating the OC and its elements into a formal ontology language, checking syntax and semantics, integrating and validating sub-ontologies, comparing and unifying conflicting terms, dissolving terminological conflicts

- *Ontology operational use* comprises publishing the OC, checking, validating its ingredients and asking for and receiving feedback, adapting the OC to super-/neighbour ontologies, looking for requests for revision, initiating revision (if necessary).

If we consider an ontology as a hierarchy of OC's, we can use the EOS model as a generalised life cycle model for developing ontologies of any size in an evolutionary way: Ontology development is then a complex process of concurrent OC developments. One particular OC may be developed in two ways: either (in a *top down* fashion) as part of the (already existing) encompassing domain ontology, or (*bottom up*) as part of a software project located in the concerned domain. In the latter case, the (ontology-related) results of the project have to be integrated into the encompassing domain ontology.

Of course, this step is not easy and is quite similar to the well-known schema integration problem. It consists of combining the separate ontology parts to a single one by identifying communalities and conflicts, while resolving the latter. However, the continuous update of domain ontologies through project ontologies allows the knowledge bases to be kept up-to-date and always relevant for further projects. Such an approach can only be successful if the domain ontologies are of high quality and if sophisticated and well-proven comparison and integration techniques are used.

4.3 Outline of an EOS-based OBSE process

With the above prerequisites, we can define an ***OBSE process*** as a combination of project and ontology development cycles. We consider a software project concerning an application domain D for which a domain ontology $\boldsymbol{O}_D$ already exists. The subset of $\boldsymbol{O}_D$ which contains all definitions and explanations relevant for our project P is defined as an ontology component OC_P. An OC development cycle may be attached to OC_P as outlined above and depicted in the left part of fig. 5.

Two so called *bridges* support the exchange of information between the domain ontology and the software project life cycles (shown on the left and right part of fig. 5, resp.). The first bridge (labelled by "*import*") is relevant in the analysis phase of the software development process. If the resulting system enters the phase of operational use and has proven stable enough, the second bridge to the ontology life cycle (called "*export*" bridge) becomes relevant.

In particular, ontology analysis of OC_P results in a glossary $\boldsymbol{GL}_P$ which can be transferred to the software project P via the *import* bridge. According to the EOS guidelines, system analysis for the project P starts from requirements which delimit the scope of the system S to be built and of its application domain D. Moreover, system analysis steps are now supported by the imported definitions of OC_P (cf. fig. 5). This way, project P profits from previous ontology work on the domain D as aimed by the overall OBSE approach.

Ontology import may also be broken down to the component structure of S. According to the EOS model, the system analysis and design steps for S lead to a component structure consisting of components X_i (I = 1, …, n). Let us suppose X_1 to be the component responsible for the application domain D. Then the analysis of X_1 implies importing the definitions of OC_P via the import bridge. If there are more components

relying on the domain D and its definitions, the same *import* procedure applies for all these components.

Following the analysis steps, the software project P goes on as prescribed by the EOS model: Components are designed, may be decomposed into sub-components and modules which run through their own development cycles. Implemented modules are tested and integrated to subsystems which in turn are integrated (in an incremental or whatever way) to form the envisaged system S.

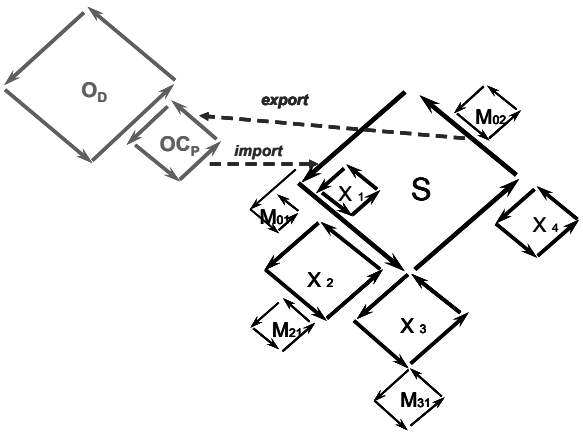

Fig. 5: System development and ontology life cycles interconnected

Operational use is the last step of every EOS development cycle – and, in particular, of the overall system development cycle (marked by "S" in fig. 5). This step is the second anchor point for OBSE-related actions: A review of the project and its results implies a particular resume of its contributions to the domain ontology. If there are any significant enhancements or modifications, these are transferred to the ontology development process of OC_P via the ***export*** bridge. Again, these contributions may be located in some component(s) of the system S, viz. in their implementations and are to be extracted via the project glossary.

5 The OBSE tool and prototype

The *OBSE tool* is intended to support software engineers who want to work along the OBSE process. The main purpose of the tool is to combine the KCPM based ontology development with the EOS software developing process. In the centre of this integration are import and export bridges (cf. fig. 5 and process description above).

- For import activities, the tool provides support by transferring elements of the domain ontology into a project knowledge base during the *analysis* phase.
- On the export side, information gained from a project is transferred to its respective domain ontology via the second bridge offered by the tool. Typically this process takes place in the *operational use* phase.

These bridges work on the glossary level. This means that the elements of the domain ontology are transferred via import functions into the KCPM glossary of a project and vice versa by export functions. However, often the knowledge to be transferred is not given in glossary form but maybe, e.g. in UML form. In order to support the transfer in these cases as well, transformations from KCPM glossaries to UML and back have been implemented [SMK 04], [Rus 07]. These transformations ensure an indirect export of UML models typically developed in software projects as well as the use of imported glossary elements in projects working with UML.

In the majority of cases both import and export requires an integration of KCPM glossary entries into existing KCPM glossaries. For example, at the beginning of a project a lot of information about the associated domain was extracted from project requirements into the project knowledge base (i.e. a conceptual model in glossary form) [M-K 02]. This model can be enhanced by elements from the domain ontology using import functionality of the OBSE tool. This is an integration process which requires specific merge functions for glossaries (cf. [V-M 05]). The integration steps must be seen as semi-automatic. Meaning the tool user can choose which elements are transferred, and specify the rules that are to be used during each integration step. The export of glossary entries into the existing domain ontology is done analogously. Since both integration functions work on the glossary level, they have been implemented in a uniform manner in the OBSE tool.

Besides the import-export functions which are essential to the OBSE process, the OBSE tool offers other features which support the management of glossaries on the domain ontology side as well as on a project knowledge base. This is not limited to graphic or table-like views of data with integrated edit function but also incorporates a built-in and always adjustable OBSE project description. This offers the user a help facility e.g. defining roles, activities and artifacts of the process, guides him/her with iteration and activity descriptions and combines the use of the tool with planning and designing activities of the process. By integrating the OBSE process description into the OBSE tool we hope to promote the ability to learn and consistently use both.

One fundamental question concerns the architecture and platform of the OBSE tool: Which architecture would best be suited for the tool having above mentioned goals in mind? For various reasons (detailed in the following) we have decided for *a PlugIn based architecture*, rooted in the *Rich Client Platform (RCP)* – at least for our first OBSE tool prototype.

RCP is a framework consisting of a relatively small core which is extendable for specific functionality via PlugIns and compatible with many operating systems.

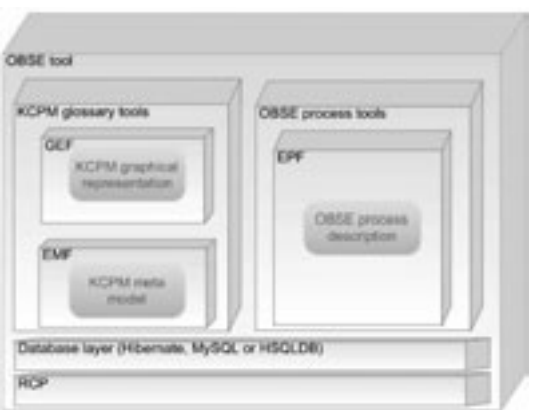

Fig. 6: OBSE tool structure

A well known implementation of RCP forms the basis of the Eclipse toolset [L-M 05]. This will be used as a platform of our prototype and allows us to construct the OBSE tools as a collection of multiple PlugIns.

Eclipse RCP elements such as perspectives and views permit the definition of different views on data for different roles and tasks in the process while maintaining uniform usage and surface. This way, different PlugIns appear to the user as a almost monolithic, homogeneous system. The *Eclipse Process Framework (EPF)* plays a key role in the OBSE tool development and the implementation is eased by its RCP based structure. It allows a description (with roles, activities, artifacts etc.) of the OBSE process to be generated and published via the tool. Another advantage of the framework is the ability to adapt process descriptions to one's own needs. Should it be necessary to define additional tasks or replace artifacts with its own variants in a project that is being carried out with OBSE this can be achieved with the help of the EPF underlying process part of the tool. This supports the scalability of the OBSE process, i.e. it makes it usable for small as well as for large projects.

We see additional advantages in other frameworks from the Eclipse Foundation. This includes the *Eclipse Modeling Framework (EMF)* and *Eclipse Graphical Framework (GEF)*. The KCPM meta model is defined with EMF. The classes of the meta models used by other PlugIns are generated by this model, including interfaces and facade classes. This provides the consistency of the KCPM meta model and the code which belongs to it. GMF is a powerful tool for implementing the graphical representations of glossary entries (cf. fig. 6). The OBSE-Tool prototype, which is currently being developed implements the above mentioned concepts and will presumably be finished by end of 2007.

6 Outlook: OBSE and MDA

Model Driven Architecture (MDA) [OMG 03] is a model centred approach which is expected to play an growing role in future SE. In the terminology of MDA three different types of models are defined: *Computation Independent Model (CIM)*, *Platform Independent Model (PIM)* and *Platform Specific Model (PSM)*. The idea is to engineer an abstract model which then can be used to generate more specific models for different target platforms. These models can be described in a modelling or natural language – note that PIM and PSM are usually expressed in UML. MDA concentrates on PIM and PSM and their transformation. For CIM only an imprecise description can be found.

We argue that project specific KCP models are suitable as CIM (cf. fig. 7). In [GDD 06] it is pointed out that CIM can be seen as some kind of ontology and as mentioned in chapter 3, the KCP method presents an appropriate way for expressing ontologies. Beyond this similarity, project specific KCP models are created from project requirements as well as from a domain ontology whereas CIMs are expected to describe the requirements for a system and the system’s immediate environment.

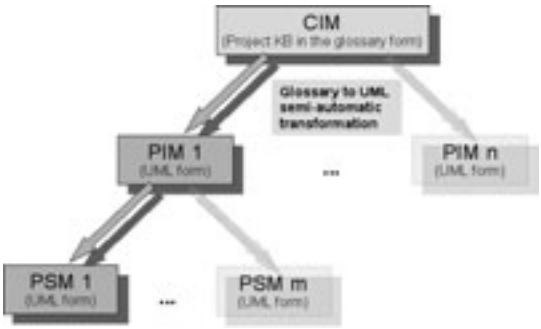

Fig. 7: OBSE and Model-driven Development (MDD)

The use of KCP models as CIM paves the way for a possible *MDA extension* going beyond the so far existing transformations which are virtually limited to the PIM and PSM stages. This extension will reduce the conceptual distance between requirements and other NL-based documents on the application domain on the one hand side and UML-like, project-specific models (PIM's) on the other. The semi-automatic mapping from KCP glossaries to UML models provides an automated transformation from CIM to PIM. It extends the MDA approach for use in the early software development phases. Moreover, our OBSE approach does not only take (project-specific) requirements into account but also (project-independent) ontologies.

This way, the scope of CIM's is extended to domain-spanning ontologies and future (mostly automated) MDA transformation chains may lead the developers from early-phase documents describing requirements and domain knowledge in glossary form through various model stages down to executable programs in some common programming language. This opens a way to combine Knowledge and Software Engineering and to make domain-specific knowledge via CIM's and glossaries reusable for professional SE projects.

References

[C-P 99] St. Cranefield and M. Purvis: A UML profile and mapping for the generation of ontology-specific content languages. In.: The Knowledge Engineering Reviews, Vol. 17.1., pp. 21-39, Cambridge Univ. Press 1999

[FKM+00] G. Fliedl, Ch. Kop, H. C. Mayr, W. Mayerthaler, Ch. Winkler: Linguistically based requirements engineering - The NIBA project. In: Data & Knowledge Engineering, Vol. 35, pp. 111 – 120 (2000)

[GDD 06] D. Gasevic, D. Djuric, V. Devedzic: Model Driven Architecture and Ontology Development. Springer 2006

[G-L 02] M. Gruninger, J. Lee: Ontology - Applications and Design. CACM 45.2, pp. 39-41 (2002)

[Gru 95] T. Gruber: Towards principles for the design of ontologies for knowledge sharing, Int. J. of Human-Computer Studies 43 (1995), also: What is an Ontology? http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[Gua 98] N. Guarino: Formal Ontology and Information Systems. In: Proc. FOIS '98, Trento (Italy), pp 3-15. IOS Press Amsterdam 1998

[Hes 96] W. Hesse: Theory and practice of the software process - a field study and its implications for project management; in: C. Montangero (Ed.): Software Process Technology, 5th European Workshop, EWSPT 96, pp. 241-256. LNCS 1149, Springer 1996

[Hes 97] W. Hesse: Improving the software process guided by the EOS model. In: Proc. SPI '97 European Conference on Software Process Improvement. Barcelona 1997

[Hes 02] W. Hesse: Das aktuelle Schlagwort: Ontologie(n). Informatik Spektrum 25.6, pp. 477-480 (2002)

[Hes 03] W. Hesse: Dinosaur Meets Archaeopteryx? or: Is there an Alternative for Rational's Unified Process? Software and Systems Modeling (SoSyM) Vol. 2. No. 4, pp. 240-247 (2003)

[Hes 05] W. Hesse: Ontologies in the Software Engineering process. In: R. Lenz et al. (Eds.): EAI 2005 - Tagungsband Workshop on Enterprise Application Integration, GITO-Verlag Berlin 2005 and: http://sunsite.informatik.rwth-aachen.de/ Publications/CEUR-WS/Vol-141/

[Hes 06] W. Hesse: Modelle - Janusköpfe der Software-Entwicklung - oder: Mit Janus von der A- zur S-Klasse. Proc. Modellierung 2006, pp. 99-114. GI-LNI P-82, Springer 2006

[HKM+ 04] W. Hesse, R. Kaschek, H.C. Mayr, B. Thalheim: Ontologien in der und für die Softwaretechnik. Proc. Modellierung 2004, Marburg, pp. 269-270. GI-LNI P-45, Springer 2004

[KFM+ 05] Ch. Kop, G. Fliedl, H.C. Mayr, M. Hölbling, Th. Horn,: Extended Tagging as a Source for Mapping Requirements Texts to Conceptual Models. In: Proc. 10th Int. Conf. on Natural Language Applications for Information Systems NLDB2005, Alicante, LNCS Springer 2005

[K-M 03] Ch. Kop, H.C. Mayr: An Interlingua based Approach to Derive State Charts form Natural Language Requirements In: Hamza M.H. (Hrsg.): Proceedings of the 7th IASTED International Conference on Software Engineering and Applications, pp. 538 – 543. ACTA Press 2003

[KMZ 04] Ch. Kop, H.C. Mayr, T. Zavinska: Using KCPM for Defining and Integrating Domain Ontologies. Proc. Int. Workshop on Fragmentation versus Integration - Perspectives of the Web Information Systems Discipline, Brisbane Australia. LNCS, Springer 2004

[KVH+05] Ch. Kop, J. Vöhringer, M. Hölbling, Th. Horn, Ch. Irrasch, H.C. Mayr: Tool Supported Extraction of Behavior Models. In: R.K. Kaschek et al. (Eds.): Proc. 4th Int. Conf. on Information Systems Technology and its Applications ISTA2005; Palmerston North (NZ), LNI Springer 2005

[L-M 05] Jean-Michel Lemieux, Jeff McAffer: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications. Addison Wesley 2005

[M-K 02] H.C. Mayr, Ch. Kop: A User Centered Approach to Requirements Modeling, Proc. Modellierung 2002, pp. 75-86. LNI p-12, Springer 2002

[Miz 98] R. Mizoguchi: Tutorial on Ontological Engineering, Osaka University 1998 http://www.ei.sanken.osaka-u.ac.jp/pub/miz/Part1-pdf2.pdf

[N-H 89] G.M. Nijssen, T.A. Halpin: Conceptual Schema and Relational Database De-sign – A fact oriented approach. Prentice Hall Publ. Comp, 1989

[OMG 03] Object Management Group (OMG): MDA Guide Version 1.0.1, http://www.omg.org/ (2003)

[Rus 07] A. Ruß: Übersetzung von UML-Diagrammen für die Ontologie-basierte Software-Entwicklung. Diploma thesis. Univ. Marburg 2007

[SMK 04] A. Salbrechter, H.C. Mayr, Ch. Kop: Mapping Pre-designed Business Process Models to UML In: Hamza M.H. (Hrsg.): Proc. of the 8th IASTED International Conference on Software Engineering and Applications, pp. 400-405. ACTA Press Cambridge (USA) 2004

[V-M 05] J. Vöhringer, H.C. Mayr: Integration of schemas on the pre-conceptual level using the KCPM-approach. Proc. 16th Int. Conference on Information Systems Development ISD2005. LNCS Springer 2005