Coupling ontology and software development processes - a rendez-vous approach

Andrej Bachmann, Wolfgang Hesse, Aaron Russ

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

Abstract. Ontology development has become a major issue in several areas of Computer Science including AI, Web technology, Knowledge Management and Information Systems. For Software Engineering (SE) projects, ontologies are particularly attractive as a means for extending the re-use principle to the early (analysis and design) phases of the development process. However, this requires a re-definition of the SE process in order to exchange knowledge with existing domain ontologies. On the other hand, ontology development is following its own process which can be defined in several ways but which has to be adapted for coupling it with the SE process.

In this article, process models for both SE and ontology development projects are highlighted and a joint, harmonised process for Ontology-based Software Engineering (OBSE) is presented. Central for this process are the uniform evolutionary process structure (based on the EOS process model), the glossary-based ontology management (following the KCP approach) and the rendez-vous principle opening the link between development cycles of both types via so-called import and export bridges. A prototype of an OBSE-supporting tool is presently being developed by our group and briefly sketched at the end of the paper.

1. Introduction

Now for about a decade, ontologies have become a major issue in several areas of Computer Science including AI, Web technology, Knowledge Management, Information System and – more recently – Software Engineering (SE). In all cases, ontologies are used wherever pieces of knowledge have to be shared – be it between automated agents performing activities in the Semantic Web, between distributed database applications performing complex queries or between software projects which share the same application domain.

For Software Engineering projects, the latter vision is particularly attractive since it might open the way for extending the re-use principle to the early (analysis and design) phases of the development process. So far, re-use has successfully been applied mainly in the late (implementation and integration) phases through re-usable components or Web services. In order to extend this principle to the earlier phases, concepts and models have to be exchanged between various projects and existing application systems. This is the anchor point for building and using (domain) ontologies in the SE field. Instead of re-defining well-known concepts and building similar (but mostly inconsistent) models for every new project in a given application domain, central
concepts are unified, stored and managed in a common domain ontology from which they can be imported to any further project located in the same domain. On the other hand, successful projects in a given domain can contribute to enhance and refine the corresponding domain ontology by integrating the gained knowledge to it. This is the core idea of our Ontology-based Software Engineering (OBSE) project.

Two important questions have to be answered before such an approach can be realised: The first question concerns the linguistic level on which domain ontologies should be defined, stored and their relevant parts be imported and merged with the concepts gained from the specific projects requirements (cf. fig. 1). For several reasons (to be further detailed below) glossaries have been chosen for this purpose following the Klagenfurt Conceptual Predesign (KCP) approach (cf. [M-K 02])

Figure 1: An ontology-based software process

The second question deals with the process model to be followed for OBSE projects. The process should comprise both software and ontology development activities and, in particular, those of exchanging knowledge between both sides. For this purpose, an evolutionary approach has been chosen based on the EOS (Evolutionary, Object-oriented Software development) model developed at Marburg university (cf. [Hes 96], [Hes 05] and below)

These two foundations of the OBSE approach are explained in the following section 2. In section 3, the OBSE process is introduced and is elaborated on its core concept – the rendez-vous mechanism. Section 4 deals with the tools for OBSE and the prototype presently being developed. In section 5, a summary is given including an outlook on the now very popular Model-Driven Development (MDD) approach and its relationship to OBSE.

The overall OBSE approach has been presented in earlier publications, e.g. [Hes 05] and [BHR+07]. In this article, we will concentrate on the process aspects of OBSE and on its core idea, the so-called rendez-vous between software and ontology development life cycles.
2. Foundations of OBSE: The KCP and EOS approaches

2.1. Glossaries as knowledge base: The KCP approach

Originally the KCP method (KCPM) was introduced as a means for supporting requirements engineering and negotiating the requirements between software developers and users [M-K 02]. The KCP language is formed of a small set of deliberately generic modelling concepts, designed for intuitive understanding.

The need for an intuitive understandable modelling language arose from the observation that while being experts in the software engineering field software developers are usually not experts for the domain the software is developed for. On the other hand, the users are domain experts but usually know little about software engineering. Consequently, domain experts often lack comprehensive understanding of complex design techniques and modelling languages. As an appropriate form to communicate requirements and domain knowledge between these groups the Klagenfurt Conceptual Pre-design language (KCPL) offers glossaries.

A small set of modelling elements allows KCP models to be displayed either in the form of easily understandable diagrams or as glossary tables. There already exist a variety of tools supporting the KCP method in general and the use of glossaries within the software engineering process in particular. On the one hand there are tools that support the semi-automatic generation of KCP models from natural text requirements [FKM+00]. And on the other hand, tools provide a semi-automatic transformation from KCP models to UML models.

As basic (static) concepts the KCP language offers modelling elements for things and modelling elements for the connections among things. These concepts are referred to as Thingtypes and Connectiontypes.

At a first glimpse, Thingtypes are very similar to UML classes and Connectiontypes very similar to UML associations. An important difference to UML is that there are no specific modelling elements for attributes. Determining whether or not a feature should be modelled as a class or as an attribute is not always clear, especially if the developer is not a domain expert. Thus instead of compelling the developer to early decide on a hierarchical order of data, he is free to keep additional information of concern in a KCP glossary – e.g. Examples for the extent of a Thingtype, cf. fig. 3 – leaving further decisions on data hierarchy to later phases in the software process.

The inter-relations between concepts can be expressed by Connectiontypes. Several Thingtypes can be involved with a Connectiontype where each Thingtype contributes its own Perspective to the relation (cf. fig. 3). The Perspective can hold additional information, such as appointing a role-name for the Thingtype within the relation or a restricting cardinality for the Thingtype. KCPM offers several more specific Connectiontypes with semantic meaning (e.g. generalization, aggregation) as well.
Figure 2: Overview of the KCP metamodel

Other KCP concepts – e.g. for modelling dynamic situations (operation types, cooperation types etc.) – are assembled similarly for use in glossaries (cf. e.g. [M-K 02a]). In the KCP metamodel, concepts are derived from the meta-class `ModelingElement`. A single concept represents a row in a glossary table. Additional information for concepts is derived from the meta-class `ModelingComponent` – represented as columns in glossaries.

### 2.2. KCPM used for Constructing Ontologies

The originally intended use for KCPM was to employ a pre-design model for gathering and communicating software project requirements which then could be used for deriving further project artefacts as e.g. a design and an implementation. But it turns out that the KCP language is suitable for describing ontologies as well [KMZ+04].

Generally, an ontology can be seen as a domain model – in contrast to a system model. Neighbors points out that “Domain analysis differs from systems analysis in that it is not concerned with the specific actions in a specific system. It is instead concerned with what actions and objects occur in all systems in an application area (problem domain)” [Nei 80]. In this regard a fundamental application of KCPM is that of a knowledge sharing mechanism for communities as is required for ontologies [Gru 93].

This is exactly the type of application KCPM has been designed for. The leaness of the language promotes a higher level of abstraction for KCP models than would be desirable for the following system design. A good example for this is the above mentioned uniform treatment of things – their distinction into classes and attributes is a matter of system design and construction rather than of domain modelling. In this respect, KCPM is similar to the Object-Role Model (ORM) [Hal 01], but its focus is more on high-level, less detailed concept descriptions and its glossary structure is particularly suited to support communication between developers and domain experts.

Furthermore, KCPM offers dynamic concepts and is supported by automated conversion tools from natural language and forth and back to UML (see above).

Ontologies can be used as a kind of template for deriving specific system models within the same domain – but for this they first need to be constructed and then maintained. The KCP method offers a tool that supports semi-automatic generation of UML models. A recently developed tool supports the automated transformation from UML models to KCP glossaries (cf. [Rus 07]) which, in general, goes along with a certain loss of
information. But this way, project experience and findings that are captured in UML
models can be exported to an existing ontology. These contributions should be seen as
tentative. After careful revision they can be integrated with this ontology yielding an
updated version of it (see also section 3.2).

The merging of conceptual models is also known as schema integration. In [V-M 05]
the advantages of the lean KCP language over other – more specific – modelling languages in this respect are discussed. Results from this research will enable extensive automated support for use of KCPM glossaries within the OBSE.

2.3. Process models for software and ontology engineering

Investigating the Software development cycle and finding paradigmatic solutions for its
dissemination and improvement has been a central issue of Software Engineering from
its origins in the late 1960-ies. Several categories of software life cycles and process
models have been published, experimented, used and (partly controversially) debated.
Outstanding paradigms are the waterfall, spiral-form, prototyping, incremental and
evolutionary models including Rational's Unified Process (RUP) as a popular instance
of the incremental paradigm (cf. [JBR 99], [Hes 03]).

Ontologies can be considered as particular artefacts having their own, often long-term
life cycle which might span the cycles of many software projects located in the same
domain. Ontological engineering (OE) has been defined as "a research methodology
which gives us a design rationale of a knowledge base kernel conceptualisation of the
world of interest, semantic constraints of concepts together with sophisticated theories
and technologies enabling accumulation of knowledge .." [Miz 98]. A systematic
discussion of the relation between Ontological Engineering and Software Engineering is
given in [FGD 02]. In a previous article, we have compared software and ontology
engineering (our preferred naming) with respect to some important criteria and have
found analogies as well as major differences (cf. [BHR+07]).

For structuring the ontology life cycle, several proposals have been made. For example,
in the METHONTOLOGY framework, the following development activities are
 distinguished: specification, conceptualisation, formalisation, implementation, mainte-
nance. (cf. [F-G 02]). Gruninger and Lee emphasise (among others) integration, sharing
and re-use activities (cf. [G-L 02]). Furthermore, at the beginning of each ontology life
cycle, we consider analysis (of the given domain or its parts) a most important task.

We agree with Fernandez et al. who have emphasised a certain analogy between the
software and the ontology development process and have investigated well-known SE
life cycle models as potential paradigms for ontology development (cf. [FGJ 97]).
Among these are:

- the waterfall model,
- incremental development, and
- evolutionary development.

Obviously, the waterfall paradigm is rather inappropriate for ontology development
since it favours a sequential procedure with stable requirements, consecutive phases,
defined intermediate results, milestones etc. – all characteristics which are normally not
given for ontology development. More promising is an incremental approach: it starts
with a kernel ontology, and proceeds with "ontology increments" i.e. formalisations of
entities of particular subdomains. But traditional incremental development has as well to follow an overall plan and is triggered by "milestones" (concerning the finalised increments) – assumptions which mostly cannot be made for ontology development.

In their comparison of OE approaches, Fernandez-Lopez et al. have found "evolving prototypes" as the most common way for establishing and enhancing ontologies [F-G 02]. This qualifies the evolutionary approach as favourite in this field. Following this approach, ontology development starts with an "ontology prototype" comprising some basic definitions – maybe stated as an initial glossary. It then consists of – usually many – development cycles concerning particular concepts or subdomains of the domain in question. A process of continuous re-development and integration is the driving motor of ontology evolution. Primarily it is driven by software projects located in that domain, i.e. each application project triggers one or more ontology evolution cycles.

2.4. EOS – a uniform, evolutionary process model

For the indicated reasons, we favour the EOS model for both the Software and Ontology Engineering processes (cf. [Hes 05]). Among its key concepts are:

- **Component-based structure**: Complex systems are viewed as hierarchical compositions of parts called components and modules. Correspondingly, an ontology can be decomposed into ontology components dealing with subdomains or with particular concepts. The issue of ontology modularisation, i.e. defining such components will be subject of a forthcoming article. Components can be developed (relatively) independent from each other but have to be integrated and adapted to the leading ontology in each development cycle.

- **"Fractal" development cycles following the system architecture**: Unlike traditional life cycle models, EOS associates development cycles with the elements of the system structure (and not vice versa): every component has its own cycle. Typically, these are performed concurrently and have to be synchronised – leading to a "fractal" process structure (cf. fig. 4, right part). Accordingly, ontology development can be viewed as a complex process structured in a "fractal" way, where single (ontology) components are elaborated separately and then integrated.

- **Homogeneous concurrent development cycles**: All development cycles (be it on the system, component or module level) have the same structure consisting of the four main phases: analysis, design, implementation and operational use. In the context of the development of an ontology component OC, these are sketched as follows (cf. fig. 3):

  - **Ontological analysis**: define the OC, delimit its boundaries, identify potential applications, analyse relevant terms and concepts, build a taxonomy, describe terms as glossary entries, and dissolve terminological conflicts.

  - **Ontology design**: define facts and rules for the OC, create visualisations, check-ups and comparisons with other glossaries, modify and (re-)structure the taxonomy and particular glossary entries, dissolve conflicts.

![Fig. 3: An EOS development cycle](image-url)
• **Ontology implementation and integration:** check syntax and semantics, integrate and validate sub-ontologies, compare and unify conflicting terms, dissolve terminological conflicts and – if needed – translate the OC and its elements to a formal ontology language,

• **Ontology operational use:** publish the OC, check, validate its ingredients, ask for and receive feedback, adapt the OC to super-/neighbour ontologies, look for revision requests and, if given, initiate these.

3. **The OBSE process**

Now we are ready to define the **OBSE process** as a combination of software project and ontology development cycles. Our main requirements on this process are:

- It should be well-suited for both software and ontology development and therefore follow an evolutionary approach (as argued above).

- It should be similar for both software and ontology development activities and should offer possibilities to synchronise corresponding development cycles and to exchange knowledge between the two sides.

- It should be component-based in that sense that components (on both the software project or on the ontology side) can be treated separately and be exchanged wherever this is necessary.

3.1. **Intertwined development cycles**

The EOS model serves as a joint framework for defining and supporting this approach. That means: Both software projects and ontology development are organised as hierarchically structured, "fractal" development cycles (cf. fig. 4). Software systems (marked by "S") are composed of components ("X") and modules ("M") which have their own development cycles depicted on the one (right hand) side and the same is true for domain ontologies (OD) with their ontology components (O DC) on the other (left hand) side.

Both development cycles are intertwined in the following way: Whenever a new software project SP starts in a given domain D which has already been captured by a given Ontology OD, its first system analysis phase (cf. the upper left black arrow in fig. 4) will comprise activities of requirements elicitation for that project in the domain D. These activities are supported and extended by importing ontology components O DC relevant for that project (cf. the dashed blue arrow).

Both activities contribute to establish and enhance the **project glossary** for SP which forms the basis for the following system design and implementation activities (cf. the lower black arrows of the system development cycle). Following the EOS principles, development cycles for system components and modules are initiated and followed-up - mostly in a concurrent way to be synchronised by EOS revision points (cf. [Hes 96], [Hes 03]).

The **system use & operations** phase (cf. the upper right back arrow) starts when a running system version is operational in its target environment. It comprises system acceptance and performance tests, configuration management, quality assurance, project evaluation and other system evolution activities. Among these activities is a **project**
glossary review with respect to possible export to the domain ontology $O_D$. All glossary entries which might be of interest for extending or modifying $O_D$ are made available for exporting them to the responsible domain experts.

![Diagram](image)

**Figure 4: System development and ontology life cycles interconnected**

### 3.2. Building import and export bridges: The rendez-vous mechanism

The import and export bridges are central parts of the infrastructure (cf. [FGD 02]) supporting the OBSE process. They include several activities and work in a semi-automatic way. Fig. 5 shows the import bridge activity flow in detail and sketches the export bridge. This bridge is built analogously to the import bridge. The only differences are the inverse data flow and different responsibilities (changes on the ontology side).

Following the KCPM approach (and its natural language-based tools, cf. [FKM+00]), all domain knowledge derivable from the requirements will be extracted and stored in the project glossary ($PG$). The elements in this glossary are step by step compared with possible occurrences of these elements in the domain ontology component local copy ($OCGLC$) – the import bridge uses a local copy in order to allow concurrent access to the domain ontology. Any occurrence of a matching element including its immediate neighbours is displayed to user. The user may identify additional elements and differences in this OCGLC extract with regard to the project glossary. These can be selected to transfer them into the PG. Then the selected elements and the already existing elements have to be adapted to each other. The result of performing these steps on all PG elements is a new (enriched) project glossary. This enrichment process can be repeated and concluded by post-editing activities.

The export bridge is used during the system use and operation phase and works in a vice versa way. It enables domain ontologies to be updated by relevant project results - this way contributing to the evolution of that ontology. There are many questions concerning ontology integration and evolution (including e.g. semantic interoperability) which have to be solved on the domain ontology side and need further discussion but cannot be dealt with in this article (for this topic see e.g. [V-M 05]).
Figure 5: Activity diagram for the Import Bridge process
4. Implementation of the rendez-vous concept: The OBSE tool

4.1. Outline of the OBSE tool

The OBSE tool offers support for different tasks of the OBSE process. Its main function is to transfer elements between different KCPM glossaries (import/export functions). Additional functionality includes creation and editing of glossaries for both – tabular and graphical – representations, conversion of KCPM glossaries into UML models and vice versa as well as the generation of KCPM glossaries from requirements. In order to support process flow management the tool contains a generalised OBSE process description. Optionally this description can be modified by project managers allowing the adaptation to individual project needs – e.g. adjusting role definitions – and offering scalability of the OBSE process to different project sizes.

Main requirements on the OBSE tool were (1) to give it a highly modular structure in order to support incremental enrichment of functionality and (2) to provide a uniform look and feel interface to the OBSE tool user. For this reason and other benefits outlined in [BHR+07] we use Eclipse Rich Client Platform (RCP) [L-M 05] designing and implementing the Tool and its components in form of PlugIns.

4.2. Tool architecture

The OBSE tool architecture shows four main components (cf. fig 6): Rich Client Platform providing basic application functionality and runtime components, Database layer packaging Database access functionality, KCPM glossary tools (PlugIns to manage KCPM glossaries) and OBSE process tools (PlugIns to guide the user through the OBSE process).

PlugIns for KCPM glossary tools have a common basis: The KCPM metamodel. For interoperability reason we use Eclipse Modeling Framework (EMF) to generate PlugIns for code implementation of the metamodel and editor functions. Additionally we combine Teneo and Hibernate\(^1\) libraries in the Database Layer thus using the metamodel for object relational mapping as well.

5. Ontologies in future software projects: From MDD to ODD?

Software Engineering and Ontology Engineering are two related but still independent areas originating from and worked out by different communities of Computer Science. They might, however, grow together as soon as the relevance and importance of ontologies for at least a certain kind of software projects is recognised. Ontologies can

---

ease and inspire software projects in their early phases of requirements elicitation and knowledge acquisition. They can help to extend the reuse principle from running code modules to glossaries and models used in the early phases to capture commonly shared definitions and domain knowledge.

Model Driven Development (MDD) is another approach being propagated as a way to promote reuse within the software development process. Starting on the model instead of the code level, MDD aims to accomplish reuse at a higher abstraction level than e.g. the older OO techniques do which target at reuse of source code or software libraries.

By the Model Driven Architecture (MDA, the core piece of MDD) three kinds of models at different abstraction layers are defined: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). The MDA approach concentrates on discussing – preferably automated – transformations from PIM to PSM. From reusable, (highly) abstract models on the PIM level, different transformations – controlled by parameters (or markers) – allow the (automatic) generation of a variety of more specific (PSM style) models. This reuse technique enables evolutionary improvement of the abstract models and thus should lead to platform-specific models built on proven quality and best practice.

While the focus of the MDA Guide [OMG 03] lies on PIM and PSM and their transformation, the meaning of CIM and its role in the MDA approach is only marginally sketched. The authors of [GDD 06] state that CIM is a domain model and in that regard has similarities to an ontology. However, in the MDA Guide, a CIM is described as a model for systems and their immediate environment thus giving it a narrower perspective than for example the domain ontology would have. This restricted notion of CIM corresponds very well to KCP glossaries that are generated from specific project requirements.

With our OBSE approach and its rendez-vous mechanisms, additional knowledge can be imported from a (KCPM-based) domain ontology into the project specific KCP glossary using integration techniques. In order to assist the task of schema integration (i.e. integrating parts of a domain ontology into a project glossary or vice versa) several procedures are researched such as pattern matching or employing heuristics for identifying similarities between the glossaries to be integrated. These and other techniques are subject to further development by Vöhringer et al. (e.g. cf. [V-M 05]).

While most adoptions of MDA ignore the transformation from CIM to PIM or assume it to be manually handled, our OBSE approach opens a good perspective to what we would like to call Ontology-Driven Development (ODD), i.e. to extend the controlled MDD transformation process up to the CIM level and to assist their initial transitions by automated, glossary-based tools.

Acknowledgement

We like to thank the reviewers for their substantial and helpful comments and our colleagues H.C. Mayr, Ch. Kop and J. Vöhringer from the KCPM group for fruitful cooperation in the OBSE project and for many inspiring discussions.
References


