52nd Internationales Wissenschaftliches Kolloquium
Technische Universität Ilmenau
10 – 13 September 2007

G. J. Nalepa / I. Wojnicki

Filling the Semantic Gaps in Systems Engineering

Abstract: The paper deals with some common problems present in knowledge and software engineering, related to the practical design, analysis, and implementation of systems. With different design methods used at subsequent design stages, the so-called semantic gaps appear, due to important differences in semantics between design methods. The paper discusses the semantic gaps present in software and knowledge engineering. In order to fill them it discusses a formal design method, based on the XTT knowledge representation.

Introduction

The paper deals with some common problems present in both knowledge and software engineering. These problems are related to the practical design, analysis, and implementation of systems in these domains. With different design methods used at subsequent design stages, the so-called *semantic gaps* appear. They are related to important differences in semantics between design methods. The best example is the problem with translating specification requirements, into a UML model, and then transforming it into object-oriented code. In order to cope with these problems some computer tools are used. However, these tools cannot solve these problems without proper formal foundations. This is why, the research in the field of formal methods in systems engineering is still active. In order to fill the semantic gaps while providing a bridge between these domains, the paper discusses a formal design method, based on the XTT knowledge representation. The method supports a hierarchical design, implementation, and on-line evaluation of systems.

Selected Issues of Software Engineering

Software engineering (SE) is a domain where a number of mature and well-proved design methods and approaches exist [1]. However, number of critical problems with efficient and integrated design and implementation of complex software persist. It will be argued, that sources of errors in software engineering are:

- *The Semantic Gap* between existing design methods, which are becoming more and more declarative, and implementation tools that remain sequential/procedural. This issue results in the problems mentioned below.
- *Evaluation problems* due to semantical differences of design methods and lack of formal knowledge model. They appear at many stages of the SE process, including the final software correctness, the validity of the design model, and the transformation from the model to the implementation.
- The so-called *Analysis Specification Gap*, which is the difficulty with proper formulation of requirements, and transformation of the requirements into an effective design, and then implementation.
- The so-called *Separation Problem*, which is the lack of separation between Core Software Logic, software interfaces and presentation layers.

The Software Engineering is derived as a set of paradigms, procedures, specifications and tools from pure programming, which is coding. Historically, when the modeled systems became more complex, SE became more and more declarative, in order to model the system in a more comprehensive way. It made the design stage independent of programming languages which resulted in number of approaches. So, while programming itself remains mostly sequential, designing becomes more declarative. The introduction of object-oriented programming does not change the situation drastically.

In software engineering the software development process and life cycle is represented by several models [1]. In this process systems analysts try to model the structure of the real-world information system in the structure of computer software system. So the structure of the software corresponds to some respect to the structure of the real-world system. The task of the programmers is to encode and implement the model in some lower-level programming language.

UML approach identifies two distant domains of Software Engineering [2]. One of them is modeling software structure the other is modeling its behavior. There are two classes of diagrams then: Structure Diagrams and Behavior Diagrams containing different types of diagrams. *Structure Diagrams* model software structure, and comply with object-oriented software engineering. It seems that Structure Diagrams are the UML basis. They are fairly complete and allow for expressing software components and denoting relationship among them easily (i.e.: Class Diagram, Component Diagram etc.). *Behavior diagrams* model software logic. It is modeled at different abstraction levels. There is a big picture perspective: modeling what particular software should do, from the user point of view (i.e.: Use Case Diagram). There is also a detailed perspective: what particular software components defined by the Structure Diagrams should do (i.e.: State Machine Diagram, etc.).

A typical software design process based on UML consists of the following stages: general behavior modeling (use cases), structure modeling, behavior and interaction modeling. The general behavior modeling describes what the system should do in the most general terms. The second stage which is structure modeling tries to describe what the system will consists of, using class diagrams mostly. But the practice indicates, that the process is in fact in most cases the know-how of the users. The fact is that, UML is only a language suitable for software design but it does not offer a design process. The process is somehow hidden, and only the final result is visible. This can be partially fixed with the methodologies such as the MDA.

Since there is no direct bridge between declarative design and sequential implementation, a substantial work is needed in order to turn a design into a running application. This problem is often referred to as a *Semantic Gap* between a design and its implementation [3]. It is worth noting, that while the conceptual design can sometimes be partially formally analyzed and evaluated, the full formal analysis is impossible in most cases. However, there is no way to assure, that even fully formally correct model, would translate to a correct code in a programming language. What is even worse, if an application is automatically generated from a designed conceptual model, then any changes in the generated code have to be synchronized with the design.

There is also another gap in the specification-design-implementation process called *Analysis Specification Gap*. It regards the difficulty with the transition from the specification to the design. Formulating a specification which is clear, concise, complete and amenable to analysis turns out to be a very complex task, even in small scale projects.

Executable Design Concept

The *executable design* concept (ED) aims at solving the main problems outlined previously. The concept itself is not new, and can be considered one of the "holy grails" of systems engineering. The main goal of this concept is to avoid semantic gaps, mainly the gap between the design and the implementation [3]. In order to do so, the following elements should be developed: a rich and expressive design method, a high-level runtime environment, and an effective design process. A full ED method should eventually shorten the development time, improve software quality, provide a design-once-run-everywhere solution, transform the "implementation" into the runtime-integration.

The development of an ED has been approached on several fronts, namely: the implementation front, with the development of new, experimental languages; as well as on the design front, with new design approaches; with a lot of recent development in the area of advanced runtimes, including virtual machines.

From the ED perspective, in the domain of software design there are at least two interesting developments. The first one concerns the extension of UML into Executable UML (xUML) with action semantics, see [3] for more details. The principal idea is to fill in gaps present in UML, in order to offer a translation from an UML specification into an executable prototype. However, it must be pointed out the the current state of the xUML is unclear, and its applications limited.

Another very important and influential concept concerns the so-called design patterns [4]. The idea is to identify certain patterns on the design level, and use them as the foundation for future design. The patterns are usually identified in the object-oriented paradigm. What is important, common patterns nowadays have practical implementations in the programming environments such as Java. So they are not only used to speedup and simplify the design, but also for providing a kind of ED.

There are a few assumptions and observations regarding ED. Since the software design process is declarative, its result, an application, is declarative as well (not counting interactions with existing non-declarative components, user interface, operating system etc.). This implies that execution of a declarative application must be provided through a declarative or at least partially declarative languages, including functional programming ones. Common choices are: Lisp, Prolog and Haskel. Moreover such an approach allows to formally analyze the designed application by the same runtime environment which runs it. It reduces number of software components implementing the runtime technology.

Finding a Bridge with Knowledge Engineering

What makes *knowledge-based systems* (KBS) distinctive is the separation of knowledge base from the knowledge processing facilities [5,6]. In order to store knowledge, KBS use various knowledge representation methods, which are *declarative* in nature. In case of rule-based systems (RBS) these are *rules*. Specific knowledge processing facilities, suitable for particular representation method being used, are selected then. In case of RBS these are logic-based inference engines.

What is important about the knowledge engineering process, is the fact that it should capture the expert knowledge and represent it in a way that is suitable for processing (this is the task for a knowledge engineer). The actual structure of a KBS does not need to be system specific - it should not "mimic" or model the structure of the real-world problem. However, the KBS should capture and contain knowledge regarding the real-world system. It should be pointed out, that in case of KBS there is no single universal engineering approach, or universal modeling method (such as

UML in SE). Different classes of KBS may require specific approaches.

It is worth considering how the standard SE language, UML, can be used to help build KBS. There are several possible approaches when it comes to practical UML application for knowledge engineering:

- Model system with a knowledge-based approach, that is use some classic knowledge representation method, such as decision trees, then design the software implementation using UML, and generate an object-oriented code.
- Model rule-based knowledge with UML diagrams, and then generate the corresponding OO code.
- Incorporate a complete rule-based logic core into an OO application, implementing I/O interfaces, including presentation layer, in an OO language.

The first solution is a “classic” and definitely the easiest one. It can be found in number of tools and approaches. In this case KE methods are used in the “design” stage, while SE methods provide “implementation” means (UML is somehow used to design the implementation previously designed with KE methods). But the fact is it can be considered the worst solution, since it exposes the semantic gap.

The second approach relies on either extending, or redefining the original semantics of UML. Some early beginning can be observed in OMG Production Rule Representation. However, a complete example of this approach may be found in the Unified Rule Modelling Language (URML) [7]. In this case existing UML diagrams are used to model different type of rules.

The last one is possibly the most complicated approach. It relies on the incorporation of the knowledge-based component into an OO application in a way that minimizes the semantic gap between SE and KE. This is the solution visible, to some extent, in the business rules approach. A similar, but more complete solution is being developed in the Hekate project, where a declarative, rule-based core is integrated into an OO application as a logical model (as in the MVC design pattern).

There are some general observations regarding the usability of UML. The syntax seems to be well defined; however, in some cases the semantics is not. One of the limitations of UML is its heavy dependability on the concept of object. This concept may be fundamental for OO languages, but it is of marginal importance for AI. The limitations of semantics are in some cases decreased with the use of UML profiles. However the problem is that in some cases profiles can totally redefine the original semantics, rendering its relation with the syntax nonexistent.

A problem is that two perspectives provided by UML (Structure and Behavior diagrams) do not mix well. While the detailed perspective corresponds to classes, the big picture one serves more as a guideline than a real modeling tool. What is worse some of the Behavior Diagrams share common functionality and judging which one to use is not clear quite often.

The semantic gap problem is the most important one from the software manufacturing perspective. Even if diagrams support the implementation process by describing software in a comprehensive way, it cannot be validated in reasonable time if the implementation matches the design. At some point there are structural diagrams which describe what the system consists of, and behavioral diagrams, describing how the system should work, and finally the implementation which consists of the designed structure and is believed to behave accordingly. It is worth noting, that while the behavior design can sometimes be partially formally analyzed and evaluated, a formal analysis of the implementation is impossible in most cases. There have been some substantial work on automating the transition from design to implementation, however none of these approaches solves the problem.

Hybrid Knowledge Engineering Methodology

The Hekate project aims at addressing the problems described previously. It is based on experiences with the Mirella project [8]. The main goal of that project was to fully develop and refine the integrated design process for RBS. The integrated design process proposed in Mirella can be considered a top-down hierarchical design methodology, based on the idea of meta-level approach to the design process. It includes three phases: conceptual, logical, and physical. It provides a clear separation of logical and physical (implementation) design phases. It offers equivalence of logical design specification and prototype implementation, and employs XTT, a hybrid knowledge representation.

Hekate [9] aims at extending Mirella's RBS perspective towards general SE. A principal idea in this approach is to model, represent, and store the logic behind the software (sometimes referred to as business logic) using advanced knowledge representation methods taken from KE. The logic is then encoded with use of a Prolog-based representation. The logical, Prolog-based core (the logic core) would be then embedded into a business application, or embedded control system. The remaining parts of the business or control applications, such as interfaces, or presentation aspects, would be developed with a classic object-oriented or procedural programming languages. The Hekate project should eventually provide a coherent runtime environment for running the combined Prolog and Java/C code.

The main idea behind *XTT* knowledge representation and design method aims at combining some of the existing approaches, namely decision trees and decision tables. It allows for a hierarchical visual representation of the decision tables linked into tree-like structure, according to the control specification provided. XTT, as a design and knowledge representation method, offers transparent, high density knowledge representation as well as a formally defined *logical*, Prolog-based interpretation, while preserving flexibility with respect to knowledge manipulation.

In Hekate, the knowledge base design process and knowledge visualization is derived from the XTT methodology. The XTT methodology is currently being extended towards covering not only forward and backward chaining RBS but also control applications, databases and general purpose software. From the implementation point of view Hekate is based on the idea of multiparadigm programming. The target application combines the logic core implemented in Prolog, with object-oriented interfaces in Java, or procedural in ANSI C. This is possible due to the existence of advanced interfaces between Prolog and other languages. Most of the contemporary Prolog implementations have well developed ANSI C interfaces. There is also a number of Object-Oriented interfaces and extensions in Prolog. The best example is LogTalk (`www.logtalk.org`).

In Hekate, the Semantic Gap problem is addressed by providing declarative design methods for the business logic. There is no translation from the formal, declarative design into the implementation language. The knowledge base is specified and encoded in the Prolog language. The logical design which specifies the knowledge base becomes an application executable by a runtime environment, combining an inference engine and classic language runtime (e.g. a JVM).

At the starting point for solving the Analysis Specification Gap problem the ARD design method is used. In Hekate Advanced Relationship Diagrams allow to specify components of the system and dependencies among them at different levels of detail. It allows to design software in a top-down fashion: starting from a very general idea what is to be designed, and going into more and more details about each single quantum of knowledge which refers to the system.

The executable design concept used in Hekate is based on ARD/XTT concept. ARD is used to describe dependencies in the knowledge base on different abstraction levels, while XTT represents the actual knowledge. The design process starts with an ARD model at a very general level, which is developed to be more and more specific. The nature of knowledge dependencies, facts and rules, are encoded with XTT. An application model based on combined XTT and ARD, along with interfaces and views, becomes the Application executed by the HeaRT (Hekate Run-Time), an inference engine supported with optional sequential (C/Java) runtime.

Two main approaches to provide an effective runtime environment for Hekate have been considered. The first one consists in generating native code in some classic object-oriented language such as Java. This solves both the practical implementation as well as runtime problem. This solution is used in products such as JBoss Rules (formerly Drools). However, it does has a major drawback: the object-oriented semantics is very distant from the declarative rule semantics of XTT. This instantly unveils a semantic gap which turns out to be a major limitation during the implementation and testing of the system.

The second approach is based on using a high-level Prolog representation of XTT. Prolog semantics includes all of the concepts present in XTT. Prolog has the advantages of flexible symbolic representation, as well as advanced meta-programming facilities. The Hekate-in-Prolog solution is based on the XTT implementation in Prolog. In this case a term-based representation is used, with an advanced meta interpreter engine provided.

Concluding Remarks

The paper discusses selected important issues present in systems engineering in the software engineering domains. These issues are commonly described as *semantic gaps* in the design process. The are responsible for making the design more complex and fragile to errors. In the paper a proposal for solving these problems is given. It is based on the idea of using knowledge engineering methods in software design. Custom representation method, the XTT, developed in the Mirella project, is the foundation of the Hekate project. Hekate aims at providing efficient AI tools for software engineering, basing on the extension of the executable design concept.

References:
[1] Sommerville I. , *Software Engineering*, Pearson Education, 2004.
[2] Fowler M., *UML Distilled: A Brief Guide to the Standard Object Modelling Language*, Addison-Wesley, 2003.
[3] Mellor S., Balcer M., *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002.
[4] Gamma E., et al., *Design Patterns*, Addison-Wesley, 1995.
[5] Torsun I. S., Foundations of Intelligent Knowledge-Based Systems, Academic Press, 1995.
[6] Ligeza A., Logical Foundations for Rule-Based Systems, Springer-Verlag, 2006.
[7] Lukichev S., Wagner G., Visual Rules Modeling, 6th International Conference Perspectives of Systems Informatics, Springer LNCS, 2006.
[8] Nalepa G. J., Ligeza A., A Visual Edition Tool for Design and Verification of Knowledge in Rule-Based Systems, Systems Science, vol. 31, num. 3, pp. 103-109., 2005.
[9] Nalepa G. J., Wojnicki I., A proposal of Hybrid Knoledge Engineering and Refinement Approach, 20th FLAIRS Conference Key West Florida, AAAI Press 2007.

Authors:
Ph. D. Eng. Grzegorz J. Nalepa
Ph. D. Eng. Igor Wojnicki
AGH University of Science and Technology
Al. Mickiewicza 30, 30-059, Krakow
Phone: 48-12-617-3856
Fax: 48-12-634-15-68
E-mail: gjn@agh.edu.pl, wojnicki@agh.edu.pl

Acknowledgemets
The paper is supported by the Hekate project founded from 2007-9 resources for science.