Using UML for Knowledge Engineering – A Critical Overview ⋆

Grzegorz J. Nalepa[1] and Igor Wojnicki[1]

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl, wojnicki@agh.edu.pl

Abstract The paper analyzes UML, the well known software engineering tool, from the knowledge engineering perspective. The goal of the paper is to evaluate UML as the possible design tool for knowledge engineering (KE). Some fundamental conceptual differences between UML and classic knowledge engineering methods are outlined. The paper aims to identify possibilities of an effective and normative UML application and extension in order to use it effectively in KE. One of the goals of the *Hekate* project is to develop such extensions, basing on the ARD and XTT knowledge representation methods.

1 Introduction

The domain of *software engineering* (SE) is driven by both research and practical industrial applications. While new technologies and applications are developed rapidly, it takes time and experience to create mature design methods and processes. In software engineering a number of well-proven methods and conceptual tools exist [1]. Today the *Unified Modelling Language* is the *de facto* standard when it comes to the practical software design. It is tailored towards modern object-oriented languages.

However, the industry seems to recognize some major limitations of UML, and UML-based methods. These limitations are related to: the UML-based design process itself, semantic gaps between different stages of design and implementation, limited capabilities of expert knowledge modeling, and problems with the so-called executable design.

This paper is written from the *knowledge engineering* (KE) perspective [2]. Being an AI domain, KE uses different conceptual tools, mainly knowledge representation methods, for knowledge modelling and processing. The research oriented towards an integration of SE and KE, has been – to some extent – always visible. Recently there have been a number of approaches towards practical incorporation of KE methods, such as decision rules-based design and implementation into SE. This trend is especially visible in the so-called *business rules approach* [3].

⋆ The paper is supported by the *HEKATE* Project funded from 2007–2009 resources for science as a research project.

In the paper some practical experiences with UML are described; these are related to a possible use of UML in the *Hekate* Project [4]. A critical evaluation of UML is given in Sect. 2, with some observations on possible UML use in KE in Sect. 3. In Sect. 4 possible applications of UML for knowledge modelling are considered. In Sect. 5 *Hekate*, Hybrid Knowledge Engineering project is described. Directions for future work are contained in the final section.

2 UML – A Critical Overview

UML approach identifies two distant domains of Software Engineering. One of them is modelling software structure, the other is modelling its behavior. There are two classes of diagrams then: Structure Diagrams and Behavior Diagrams containing different types of diagrams.

Structure Diagrams, which model software structure, 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 modelled at different abstraction levels. First of all there is a big picture perspective: modelling 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, Interaction Diagram etc.). And the problem is that these two perspectives do not mix well with the Structure Diagrams. While the detailed perspective corresponds to classes, the big picture one serves more as a guideline, than a real modelling tool. What is even worse, some of the Behavior Diagrams share common functionality, and judging which one to use is often not clear.

It seems that there is almost no relationship between modelling software behavior and its structure. If the structure is guessed, then the behavior can be added, but it seems to be a really hard task to infer the structure from the behavior.

There is another issue with interpretation of the diagrams, either over interpretation or under interpretation. The diagrams tend to be not clear, and some additional explanation describing diagram semantics is needed. To deal with this issue, *UML Profiles* have been introduced. They tend to specify meaning of the diagrams regarding particular application or domain. While they offer a solution for diagram interpretation, they do not deal with the modelling software behavior, and subsequently its structure, as it was pointed out earlier.

A typical software design process based on UML consists of the following stages: general behavior modeling (use cases), structure modeling, behavior and interaction modeling [5]. The general behavior modeling (applying Use Case Diagrams) 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. And here comes an issue. While knowing

what the system should do one has to guess *what the system consists of* (with use of Class Diagrams). There is no consistent and clear process, or guidelines, regarding the transition from the use case diagrams to class diagrams. Coming up with class diagram is often an iterative process of trail-and-error kind. It involves a strong feedback from other stages. Often, while having a preliminary class diagrams, subsequent behavioral and interaction diagrams are created, then class diagrams are reviewed and updated, behavioral and interaction are updated and so on. This process is not recorded and it is informal – as a result there are ready to use, but highly complicated structure diagrams. It is usually hard to perceive from these diagrams what the system should do and how, not mentioning, that any modification requires deep analysis.

There are two more issues which are not covered by UML. The first one regards the *semantic gap* between the design and the implementation [6]. The second one regards the gap between the specification and the design and is called the *analysis specification gap*.

From the software manufacturing perspective, the *semantic gap* problem is the most important one. Having a well designed piece of software, at some point there is a need to implement it. Even if discussed diagrams support the implementation process by describing software in a comprehensive way, it is impossible to verify in a reasonable time if the implementation matches the design. At some point, there are structural diagrams which describe what the system consists of, behavioral diagrams, describing how the system should work, and finally the implementation, in case of which, it is believed that it corresponds to the designed structure and that it behaves 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 conducted to automate a transition from design to implementation, however none of these approaches solves the problem. It is a well know UML disadvantage which is called *only the code is in sync with the code* [7]. An application which is automatically generated from a designed behavioral and structural model, is not 100% complete. Some parts are missing which have to be coded by hand. And here comes an issue. Often, such automatically generated code is barely human readable. An attempt to modify it, to make complete software, involves a tremendous amount of time. What is even worse, applying such changes by hand can make the implementation not compliant with the design. In general, the design tends to be declarative, while implementation is sequential, and far from being declarative. There is a lack of compatibility between these two separate approaches then, which constitutes the mentioned earlier *semantic gap*.

There is also another gap in the specification-design-implementation process called *analysis specification gap* [8]. It regards a difficulty with the transition from a 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. While use cases try to model user requirements, there is actually no guarantee that the structure model complies with them.

Assuming that use cases describe exactly what user wants the software to do, there is still no certainty that the system structure is capable of fulfilling the cases.

3 Observations

There are some general observations regarding the usability of UML. As a language it has a syntax, semantics, pragmatics. 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 an *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.

In case of the UML language pragmatics is about the process of using it, in this case the UML-based design process. But the practise indicates, that the process is 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, which tries to formalize the process.

Another domain where important problems of UML are exposed is the so-called *executable design*. Shortly speaking, in the field of SE, it aims at providing design methods that could translate directly into an executable. The main solution concerns the extension of *UML* into the *Executable UML* (*xUML)* with *action semantics* (see [9] 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 that the current state of the *xUML* is unclear, and its applications are limited.

The two gaps described in Sec. 2 make UML design approach unreliable. The unreliability is present at the very early stage which is the *analysis specification gap*, and later on between the design and the implementation, which is the *semantic gap*. Extending UML is not easy – introducing new diagrams is somehow prohibited. It seems that OMG tends to assume that thirteen diagrams, that UML currently consists of, is a finite and complete set, which does not need to be extended any more. Some functionality of the diagrams still overlaps with other diagrams.

The following observations regarding Knowledge Engineering with UML may be formulated. Applying UML as a Knowledge Engineering method is not straight forward. Existing diagrams are not suitable for rule modeling or expressing knowledge in general. Using an UML profile, which is redefining the semantics of certain diagrams, does not help much, and in some cases might complicate the design. It forces using existing diagrams for purposes they were not designed for i.e. representing rules is tricky and inefficient. OMG request for proposal [10] tries to adapt Action Diagrams for modeling rules. Actual rules are textual and

they are fired upon transitions. It is not clearly defined what rules operate on or whether other actions in addition to the transitions are allowed or not.

4 UML Applications for Knowledge Engineering

There are several possible approaches when it comes to practical UML application for knowledge engineering:

1. Model system with a knowledge-based approach, that uses some classic knowledge representation method, such as decision trees, then design the software implementation using UML, and generate an object-oriented (OO) code.
2. Model rule-based knowledge with UML diagrams, and then generate the corresponding OO code.
3. 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 the "classic" and definitely the easiest one. It can be found in a 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 so-called *semantic gap* [11]. The problem is, that there is a fundamental difference in the semantics of the KE methods, such as decision rules, and UML.

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* [12], where some ideas of extending existing semantics of UML were contained. However, a complete example of this approach may be found in the *Unified Rule Modelling Language* (URML), (see [13]). 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 [3]. 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 Model-View-Controller design pattern [14]).

5 The Hekate Approach

The approach considered in this paper is based on an extended rule-based model. The model uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems (RBS).

5.1 Knowledge Representation

The *XTT* (*EXtended Tabular Trees*) knowledge representation [15], has been proposed in order to solve some common design, analysis and implementation problems present in RBS. In this method three important representation levels has been addressed:

- *visual* – the model is represented by a hierarchical structure of linked extended decision tables,
- *logical* – tables correspond to sequences of extended decision rules,
- *implementation* – rules are processed using a Prolog representation.

On the visual level the model is composed of extended decision tables. A single table is presented in Fig. 1. The table represents a set of rules, having the same attributes. A rule can be read as follows:

$(A1 \in a11) \wedge \ldots \wedge (An \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1).$

It includes two main extensions compared to classic RBS: 1) non-atomic attribute values, used both in conditions and decisions, 2) non-monotonic reasoning support, with dynamic assert, retract operations in decision part. Every table row correspond to a decision rule. Rows are interpreted from top row to the bottom one. Tables can be linked in a graph-like structure. A link is followed when rule (row) is fired.

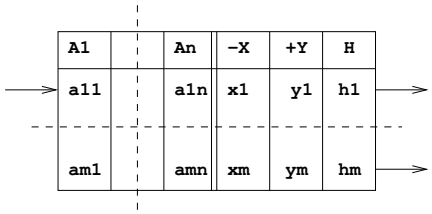

Figure 1. A single XTT table

On the logical level, a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table. The rule is based on a *attributive language* [16,17]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p is a literal in SAL (set attributive logic) (see [16]) in a form $A_i(o) \in t$ where $o \in O$ is a object referenced in the system, and $A_i \in A$ is a selected attribute of this object (property), $t \subseteq D_i$ is a subset of attribute domain A_i. Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using Prolog-like assert/retract operators in rule decision. Rules are implemented using Prolog-based representation (see [18]). Rule representation uses Prolog terms, which is a very flexible solution. However, it requires a dedicated meta-interpreter [19].

Attempts to represent XTT diagrams with UML failed. None of thirteen available UML diagrams serves the purpose of rule modeling with similar expressiveness as XTT. The best candidates were: the State Machine Diagram and the Activity Diagram. While it is possible to express rules with them, and explicitly define system states, such diagrams tend to grow very fast while software is being developed. It turned, out that these diagrams can be effectively used for small scale cases, only with a few rules. They are not suitable for real life rule-based systems.

5.2 The Design Process

In addition to XTT which represents rules, there is an entire design process involved. XTT diagrams are at the very end of this process. It is an *Attribute Relationship Diagram* (ARD) based approach. It offers a process of identifying attributes and relationships among them. Attributes are subsequently identified at more and more detailed levels. The process includes all levels. At the most detailed level, XTT diagrams are added to precisely define dependencies among attributes and to describe how to calculate attribute values.

The key underlying assumption in the ARD design with knowledge specification in attributive logics is that, similarly as in the case of Relational Databases [20], the attributes are *functionally dependent*. A basic ARD table for specification of such a functional dependency is presented in Fig. 2. The attributes on the left (i.e. the ones of X) are the independent ones, while the ones on the right (the ones of Y) are the dependent ones. An ARD *diagram* is a conceptual system model at a certain abstract level. It is composed of one or several ARD *tables*. If there are more than one ARD table, a partial order relation among the tables is represented with *arcs*. The ARD model is also a hierarchical model. The most abstract level 0 diagram shows the functional dependency of *input* and *output* system attributes. Lower level diagrams are less abstract. An ARD diagram of level i can be further *transformed* into a diagram of level $i+1$, which is more detailed (specific). A transformation includes *table expansion* and/or *attribute specification*.

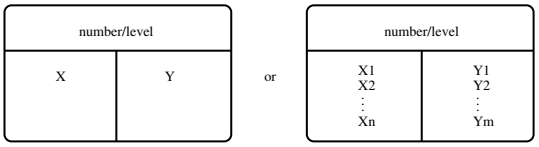

Figure 2. An ARD table: the basic scheme for $X \rightarrow Y$

At first sight, the ARD process is similar, in terms of its goals, to Structure Diagrams. However, while the Structure Diagrams tend to describe what elements the software consists of, ARD describes what is *known* about it.

5.3 Hekate and UML

The main difference between the Hekate knowledge representations and UML diagram is, that UML, after all, does not provide *a design process*. Whereas, Hekate *is* about the integrated design process. So the methods on which Hekate is based, have been invented with the design *process* in mind. Where UML provides means to model the system from different perspectives, or aspects; Hekate focuses on the subsequent phases of the design process of the very same system. So different levels of abstraction (e.g. subsequent ARD levels, or XTT design iterations) describe *the very same holistic model* all the time.

It could be argued, that this comparison is not correct, simply because UML is not the design *methodology*, but merely a *language*, providing an alphabet (diagrams), syntax (structures) and semantics (the default diagram interpretation in terms of the object-oriented programming). So a more accurate comparison of Hekate is with the MDA.

What makes the integration of UML with knowledge-based approach difficult, is the very strong assumption behind UML, that "the world can be accurately described in terms of so-called objects". Knowledge engineering is much more abstract and flexible with a number of different symbolic knowledge representations, objects being just one of them (object-oriented methods where anticipated with classic *frames* by M. Minsky.).

6 Future Work

UML seems to be a poor tool for modelling KBS for now. However, there is a way out. Research in this domain can go two ways. Either the UML Rule Profile evolves into more precise and detailed specification, or current thirteen diagrams are extended.

It could have been concluded that applying UML in the domain of knowledge engineering is possible but not worth the effort, especially if there are already well developed Knowledge Engineering methodologies present. Or perhaps expressiveness of UML is just not sufficient to cover this domain? It seems that there is no consistent answer to that. One way or the other, research in this domain could be fruitful, especially detailed specification of rule modelling with the existing Behavior Diagrams. Having such a specification better defined than that of [12], could lead to applying UML in the KE which can make it truly Unified Modelling Language.

The other possible course of action could be extending current UML diagrams with knowledge oriented ones such as ARD/XTT. While having these thirteen diagrams for general software modelling, additional ARD and XTT diagrams would provide Behavior Modelling at the level not covered by UML so far. The OMG is reluctant to extend the current diagrams but applying ARD and XTT to UML is still subject of further research.

The research presented in this paper is work in progress. Investigating possible applications of UML in the knowledge engineering domain is an important

issue in the Hekate project. The project should eventually provide UML improvements, or methods superior to UML.

References

1. Sommerville, I.: Software Engineering. 7th edn. International Computer Science. Pearson Education Limited (2004)
2. Torsun, I.S.: Foundations of Intelligent Knowledge-Based Systems. Academic Press, London, San Diego, New York, Boston, Sydney, Tokyo, Toronto (1995)
3. Ross, R.G.: Principles of the Business Rule Approach. 1 edn. Addison-Wesley Professional (2003)
4. Nalepa, G.J., Wojnicki, I.: A proposal of hybrid knowledge engineering and refinement approach. In Sutcliffe, G.C.J., Goebel, R.G., eds.: FLAIRS 2007 : proceedings of the 20th international Florida Artificial Intelligence Research Society conference, AAAI Press (2007)
5. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edn. Addison-Wesley Professional (2003)
6. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
7. Reeves, J.W.: Code as design. developer.* The Independent Magazine for Software Professionals (1992,2005)
8. Rash, J.L., Hinchey, M.G., Rouff, C.A., Gracanin, D., Erickson, J.: A tool for requirements-based programming. In: Integrated Design and Process Technology, IDPT-2005, Society for Design and Process Science (2005)
9. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. 1st edn. Addison-Wesley Professional (2002)
10. OMG: Production rule representation rfp. Technical report, Object Management Group (2003)
11. Merrit, D.: Best practices for rule-based application development. Microsoft Architects JOURNAL **1** (2004)
12. OMG: Production rule representation. Technical report, Object Management Group (br/2003-09-03)
13. Lukichev, S., Wagner, G.: Visual rules modeling. In: Sixth International Andrei Ershov Memorial Conference Perspectives Of System Informatics, Novosibirsk, Russia, June 2006. LNCS, Springer (2005)
14. Burbeck, S.: Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign (1992)
15. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31**(2) (2005) 89–95
16. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
17. Ligęza, A., Fuster-Parra, P.: A granular attribute logic for rule-based systems management within extended tabular trees. In Trappl, R., ed.: Cybernetics and systems 2006 : proceedings of the eighteenth European meeting on Cybernetics and systems research. Volume 2., Vienna, Austrian Society for Cybernatic Studies (2006) 761–766

18. Nalepa, G.J., Ligęza, A.: Prolog-based analysis of tabular rule-based systems with the xtt approach. In Sutcliffe, G.C.J., Goebel, R.G., eds.: FLAIRS 2006: proceedings of the 19th international Florida Artificial Intelligence Research Society conference, AAAI Press (2006) 426–431
19. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
20. Connolly, T., Begg, C., Strechan, A.: Database Systems, A Practical Approach to Design, Implementation, and Management. 2nd edn. Addison-Wesley (1999)