A Proposal of Hybrid Knowledge Engineering and Refinement Approach

Grzegorz J. Nalepa and **Igor Wojnicki**
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 deals with some possible applications of Knowledge Engineering methods to the practical Software Engineering. Such an integration could provide a way to overcome some constant problems present in the contemporary Software Engineering. This paper describes foundations of the HEKATE Project, which aims at incorporating well established Knowledge Engineering tools and paradigms into the Software Engineering domain. A new Hybrid Knowledge Engineering (HEKATE) methodology is proposed, which would allow for faster development of highly reliable software. The HEKATE design process is introduced which offers important capabilities of formal verification, and gradual refinement of software from the conceptual model stage to an executable prototype. An integrated design environment and runtime based on the ARD/XTT concept is also proposed. Furthermore, HEKATE-based applications can be integrated with existing software, designed in a classical way.

Introduction

Knowledge-based systems (KBS) are an important class of intelligent systems originating from the field of Artificial Intelligence (Russell & Norvig 2003). They can be especially useful for solving complex problems in cases where purely algorithmic or mathematical solutions are either unknown or demonstrably inefficient. In AI, *rules* are probably the most popular choice for building knowledge-based systems, that is the so-called rule-based expert systems (Jackson 1999; Ligęza 2006). Rule-based systems (RBS) constitute today one of the most important classes of KBS. Building real-life KBS is a complex task. Since their architecture is fundamentally different from classic software, typical Software Engineering approaches cannot be applied efficiently. Some specific development methodologies, commonly referred to as *Knowledge Engineering* (KE), are required.

Software Engineering (SE) does not contribute much concepts in Knowledge Engineering. However, it does provide some important tools and techniques for it. On the other hand some important Knowledge Engineering conceptual achievements and methodologies can be successfully transferred and applied in the domain of Software Engineering (Maurer & Ruhe 2004). This conclusion is drawn based on several observations related to the Software and Knowledge Engineering concepts described below.

This paper presents a concept of a new software engineering approach based on knowledge engineering methods. In the paper some important features of both KE and SE approaches are summarized. Furthermore, common design and evaluation problems encountered in SE are outlined. Most of these problems can be successfully approached, and possibly minimized in the field of KE and RBS design. This is why, selected concepts and tools developed for the MIRELLA Project are presented; they aim at supporting the design and evaluation of RBS. Finally, the main concept of the *hybrid knowledge engineering*, on which the HEKATE project is based, is discussed. The paper ends with concluding remarks where some main features of HEKATE are summarized.

From Knowledge to Software Engineering

It is asserted, that some important concepts and experiences in the field of Knowledge Engineering could be transferable into the domain of Software Engineering. Several observations regarding relations between these two approaches are discussed below.

Knowledge Engineering Approach

What makes KBS distinctive is the separation of knowledge storage (the knowledge base) from the knowledge processing facilities. In order to store knowledge, KBS use various knowledge representation methods, which are *declarative* in nature. In case of RBS these are *production 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.

The knowledge engineering process, in case of RBS, involves two main tasks: knowledge base design, and inference engine implementation. Furthermore, some other tasks are also required, such as: knowledge base analysis and verification, and inference engine optimization. The performance of a complete RBS should be *evaluated* and *validated*. While this process is specific to expert systems, it is usually similar in case of other KBS.

What is important about the 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. The task of programmers is to develop processing facilities for the knowledge representation. The level at which KE should operate is often referred to as *the knowledge level* (Newell 1982).

It should be pointed out, that in case of KBS there is no single universal engineering approach, or universal modelling method (such as UML in software engineering). Different classes of KBS may require specific approaches, see (Ligęza 2006; Torsun 1995).

Having outlined the main aspects of KBS development, it can be discussed how they are related to classic software engineering methods.

Software Engineering Approach

Software engineering (SE) is a domain where a number of mature and well-proved design methods exist; furthermore, the software development process and its life cycle is represented by several models. One of the most common models is called *the waterfall model* (Sommerville 2004). In this process a number of development roles can be identified: users and/or domain experts, system analysts, programmers, testers, integrators, and end users (customers). What makes this process different from knowledge engineering is the fact that 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 extent, to the structure of the real-world system. The task of the programmers is to encode and implement the model (which is the result of the system analysis) in some lower-level programming language.

The most important difference between software and knowledge engineering, is that the former tries to model how the system works, while the latter tries to capture and represent what is known about the system. The knowledge engineering approach assumes that information about how the system works can be inferred automatically from what is known about the system.

Common Design and Evaluation Problems

Having outlined some distinctive features of KE and SE approaches, several observations can be made in the field of Software Engineering. They provide a basis for a critical overview of current SE approaches and pinpointing most common problems. These issues are presented below.

Observations

Historically, there has always been a strong feedback between SE and computer programming tools. At the same time, these tools have been strongly determined by the actual architecture of computers themselves. For a number of years there has been a clear trend for the software engineering to become as implementation-independent as possible. Modern software engineering approaches tend to be abstract and conceptual (Sommerville 2004).

On the other hand, knowledge engineering approaches have always been device and implementation-agnostic. The actual implementation of KBS has been based on some high level programming languages such as Lisp or Prolog. However, modern knowledge engineering tools heavily depend on some common development tools and programming languages, especially when it comes to user interfaces, network communication, etc.

It could be said, that these days software engineering becomes more knowledge-based, while knowledge engineering is more about software engineering. This opens multiple opportunities for both approaches to improve and benefit. Software engineering could adopt from knowledge engineering advanced conceptual tools, such as declarative knowledge representation methods, knowledge transformation techniques based on existing inference strategies, as well as verification, validation and refinement methods.

This trend is already visible in the *Model-Driven Architecture* proposed by OMG (Miller & Mukerji 2003). It is a new software engineering paradigm that tries to provide a unified design and implementation method and appropriate tools for the declarative specification of software. MDA has already been adapted for business logic applications, so-called *business rules* approach (Ross 2003; von Halle 2001).

In order to improve and better integrate KBS with existing software, knowledge engineering could adopt programming interfaces to existing software systems and tools, interfaces to advanced storage facilities such as databases and data warehouses, modern user interfaces, including graphical and web-based ones. In this paper a concept of possible integration of some KE solutions with SE is put forward.

Critical Overview

The Software Engineering is derived as a set of paradigms, procedures, specifications and tools from pure programming. It is heavily tainted with the way how programs work which is the sequential approach, based on the Turing Machine concept. Historically, when the modelled 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; the best example is the MDA approach (Miller & Mukerji 2003). So, while programming itself remains mostly sequential, designing becomes more declarative. The introduction of object-oriented programming does not change the situation drastically. However, it does provide several useful concepts, which simplify the coding process.

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 (Mellor & Balcer 2002).

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. The exceptions include purely formal design methods, such as Petri Nets, or Process Algebras. However, there is no way to as-

sure, 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. It is not always possible because of the lack of compatibility between these two separate approaches: the declarative model and sequential application, which constitutes the mentioned earlier *semantic gap*. Sometimes such a code is generated in a way, which is barely human readable.

There is also another gap in the specification-design-implementation process called *Analysis Specification Gap* (Rash *et al.* 2005). 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.

Problem statement

It could be summarized, that constant 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 large differences in semantics of design methods and lack of *formal* knowledge model. They appear at many stages of the SE process, including not just the correctness of the final software, but also 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.

While some some of the methodologies, (mainly the *MDA*) and design approaches (mainly the *MVC* (Model-View-Controller) (Burbeck 1992)) try to address these issues, it is clear that they do not solve the problems. However, it seems that some of the problems could be more easily solved in case of RBS, thanks to the fact that the field is narrower and well formalized. The proof of concept is the MIRELLA (Nalepa 2004) approach which is shortly discussed below. Within this approach a new knowledge representation method and design process has been developed. Based on outcomes from the MIRELLA Project, a foundation of a more general approach to software design, called *HeKatE*, is proposed.

Mirella Project

In (Nalepa 2004) results of a research and evaluation of multiple design and evaluation methods for RBS have been presented. The main contribution of (Nalepa 2004) was: the XTT (*Extended Tabular-Trees*) knowledge representation method, the concept of an integrated design process for RBS, and a prototype Mirella CASE tool. Further developments include ARD (*Attribute-Relationship Diagrams*) conceptual design (Nalepa & Ligęza 2005; Ligęza 2006). These results are a basis for the MIRELLA Project, (see `mirella.ia.agh.edu.pl`). The main goal of the project is to fully develop and refine the integrated design process for RBS. All of these are shortly introduced below.

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. The methodology is supported by a CASE tool.

The main goal of the methodology is to move the design procedure to a more abstract, logical level, where knowledge specification is based on the use of abstract rule representation. The design specification can be automatically translated into a low-level code, including Prolog and XML, so that the designer can focus on logical specification of safety and reliability. On the other hand, selected formal system properties can be automatically analyzed on-line during the design, so that its characteristics are preserved. The generated Prolog code constitutes a prototype implementation of the system. Since it is equivalent to the visual design specification it can be considered an executable.

The main idea behind *XTT* (Nalepa 2004) knowledge representation and design method aims at combining some of the existing approaches, namely decision trees and decision tables, by building a special hierarchy of Object-Attribute-Tables (Ligęza, Wojnicki, & Nalepa 2001; Ligęza 2006). It allows for a hierarchical visual representation of the OAV 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. On the *machine readable level* XTT can be represented in an XML-based markup language, and possibly be translated to other XML-based rule markup formats such as *RuleML*.

The *conceptual design* of the RBS aims at modelling the most important features of the system, i.e. attributes and functional dependencies among them. ARD stands for *Attribute-Relationship Diagrams* (Nalepa & Ligęza 2005; Ligęza 2006). It allows for specification of functional dependencies of system attributes using a visual representation. 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, i.e. they are close to full system specification. They contain

also some intermediate conceptual variables and attributes.

The *eXtended Tabular Trees*-based design method introduces possibility of on-line system properties analysis and verification, during the system design phase. Using XTT as a core, an *integrated design process*, covering the following phases has been presented in (Nalepa 2004):

1. *Conceptual modeling*, in which system attributes and their functional relationships are identified; during this design phase the ARD modelling method is used. It allows for specification of functional dependencies of system attributes using a visual representation. Using this model the logical XTT structure can be designed.
2. *Logical design with on-line verification*, during which system structure is represented as XTT hierarchy, which can be instantly analyzed, verified (and corrected, if necessary) and even optimized on-line, using Prolog.
3. *Physical design*, in which a preliminary Prolog-based *implementation* is carried out. A RuleML translation of the XTT rule base is also available.

Using the predefined XTT translation it is possible to automatically build a prototype. It uses Prolog-based metalanguage for representing XTT knowledge base and rule inference (also referred to as XTT-PROLOG).

A prototype CASE tool for the XTT method called *Mirella Designer* (Nalepa 2004) has been developed. It supports XTT-based visual design methodology, with an integrated, incremental design and implementation process, providing the possibility of the on-line, incremental, verification of formal properties. Logical specification is directly translated into Prolog-based representation providing an executable prototype, so that system operational semantics is well-defined.

The approach is based on the idea of a knowledge representation method which offers the *design and implementation equivalence* by a direct XTT $\rightarrow$ Prolog mapping. Using a visual design method the designer can focus on building the system structure, since the prototype implementation can be *dynamically generated* and *automatically analyzed*. The approach discussed herein offers strict, formal description of system attributes and structure, creates a framework for integrating the design and verification process, and supports the design and verification process by a CASE tool. In this way, it is possible to assure that some safety-critical system properties such as attribute domains, and basic system structural logical constraints are preserved during the design process.

HeKatE – Hybrid Knowledge Engineering

The HEKATE project addresses the problems described previously. It is based on experiences with the MIRELLA project but it extends its RBS perspective towards 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 such as Java or C. The HEKATE project should eventually provide a coherent runtime environment for running the combined Prolog and Java/C code.

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* (de Moura 2003) (`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). It is called an *Executable Design* (ED) concept.

The knowledge base design process and knowledge visualization is derived from the XTT methodology. The XTT methodology is currently being extended (code name XTT^2) towards covering not only forward and backward chaining RBS but also control applications, databases and general purpose software.

At the starting point for solving the *Analysis Specification Gap* problem the ARD method is used. In HEKATE, ARD is extended and renamed to *Advanced Relationship Diagrams*. ARD allows 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. This approach is somehow similar to the *Requirements Based Programming* proposal, however implemented in a different way that this of R2D2C (Rash *et al.* 2005).

The *executable design* concept is presented in Fig. 1. It is based on ARD/XTT concept. ARD is used to describe dependencies in the knowledge base on different abstraction levels, while XTT^2 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^2. An application model based on combined XTT^2 and ARD, along with interfaces and views, becomes the Application. The Application, in turn, is executed by the *HeaRT* (*Hekate Run-Time*), an inference engine supported with optional sequential (C/Java) runtime.

The HEKATE project provides means for the design and implementation of software logic, and the integration of this logic with the presentation layer, which is in some cases

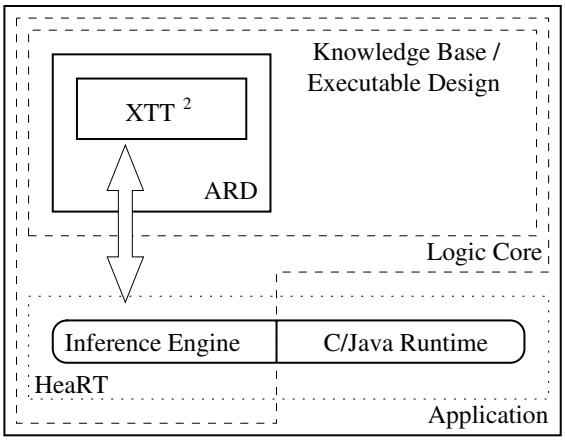

Figure 1: *Executable Design* Concept.

minimal, or even optional. It would allow for integrating and interfacing the *Executable Design* with existing technologies i.e. interfaces written using classical sequential ways provided by object-oriented or procedural languages. It would be also possible to interface with existing modules and libraries implemented in procedural (or object-oriented) languages – they often provide access to specialized hardware, or communication protocols. The approach forms so-called *Multiparadigm Programming*, making a bridge between declarative logic and sequential presentation (this is where „hybrid" comes from). It is worth pointing out that this is not to negate a possibility of declarative presentation layer design, but to provide compatibility with other, conservative programming approaches. A declarative presentation layer is also a research thread within HEKATE. Regardless, whether the design contains a presentation layer or not, there is a clear separation between it and the software logic.

In some aspects, there is a analogy between some solutions within *HeKatE* and the MVC approach used in object-oriented software designs. It consists in strong separation between the software logic model, and the presentation layer. However, in *HeKatE* the emphasis is on the rich *formally* designed and analyzed knowledge-based model.

At first sight, the *HeKatE* point-of-view may seem somehow similar to the MDA approach, and the formalized transition from the PIM to the PSM. However, in *HeKatE* different abstraction layers correspond to different levels of the knowledge base specification (more, and more detailed). No different implementation technologies are considered, since the Prolog-based unified run-time environment is provided by *HeaRT*.

The above multiparadigm hybrid approach is presented in Fig. 2. The application's logic is given in a declarative way as the Knowledge Base. Interfaces with other systems (including Human-Computer Interaction) can be provided in classical sequential manner. There is a bridging module between the Knowledge Base and the sequential Code (C/Java language code): the *Sequential Language Interface* (SLIN). It allows communication in both directions. The Knowledge Base can be extended, new facts or rules added by a stimuli coming through SLIN from the View/Interface. There are two types of information passed this way: events generated by the HeaRT Runtime and knowledge generated by the Code. Any inferred knowledge, facts or even rules could be passed to other systems, or visualized by the View/Interface through SLIN.

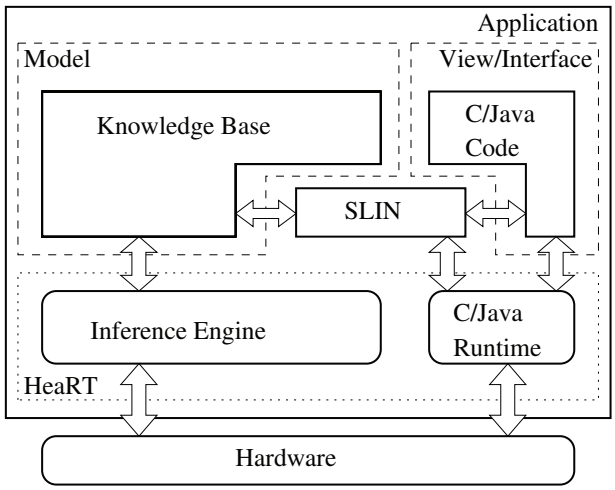

Figure 2: Multiparadigm Hybrid Approach: an Application.

It is hoped, that this methodology could provide universal modelling methods for software design and implementation. The HEKATE project aims at applying this methodology to practical design and analysis of real-life software. Main goals of the HEKATE project are to: develop an extended, hierarchical methodology for practical design, analysis and implementation of selected software classes, build CASE tools package supporting this methodology, test the approach on illustrative software examples, and benchmark test cases.

The projects focuses on wide class of software, namely two very different "benchmark" classes, that is: general business software based on the so called *business logic*, (including business rules), and low-level control software (possibly for the embedded control systems, based on a *control logic* (Nalepa & Zięcik 2006)). Other software classes, such as general purpose or scientific software are also considered.

HEKATE is currently (fall 2006) in a very early development stage. See the project webpage at `hekate.ia.agh.edu.pl` for more up to date information on the project progress, tools and technologies.

Evaluation and Refinement Issues

It is important to emphasize, that compared to some standard software engineering approaches, in *HeKatE* there are no differences in semantics of design methods. Thanks to the XTT-based logic core, the knowledge base is described using a *formal* knowledge model. This allows for avoiding some common *evaluation problems*. This also opens up possibilities of formal analysis, including verification and evaluation. Such an analysis can be provided *at the design stage*, which in turn allows for gradual *refinement* of the designed

system. In HEKATE, this aspect is referred to as *EVVA*, that is Evaluation, Verification, Validation and Property Checking of the designed KB. This approach makes software testing stage shorter and the bug squashing process becomes mostly non-existent. Important properties of the future application can be validated in the design stage and it is guaranteed that they will remain valid during execution because of the nature of the *Executable Design*.

Concluding Remarks

The paper presents a concept of a hybrid design methodology with multiparadigm approach to the software implementation. This concept is being developed within the HEKATE project. It offers superior capabilities of formal verification, and gradual refinement of the system from the conceptual model phase to an executable prototype. This is possible due to: XTT^2, ARD, knowledge representation methods and Prolog-based implementation.

The project will deliver the following software components:

- an integrated ARD/XTT^2 Design Environment,
- the Inference Engine, being a run-time environment for applications,
- the Sequential Language Interface, providing an interface with other (sequential) programming languages and environments.

There are the following main features of the proposed *Hybrid Knowledge Engineering* approach regarding SE:

- *consistency*: the *Semantic Gap* in the design process is decreased or even eliminated,
- *reduced implementation time*: the design becomes an application; enabled by the *Executable Design* concept, the implementation time of the business logic is almost zeroed,
- *prone to errors*: Evaluation, Verification, Validation and Property Checking (EVVA) is provided in the design stage by the integrated ARD/XTT^2 design environment,
- *prone to bugs*: since the application design is validated and the design is simultaneously implementation (thanks to the *Executable Design* concept) the programming bugs can be eliminated.

It is believed that ultimately, the proposed methodology is going to provide an alternative for contemporary Software Engineering approaches.

References

Burbeck, S. 1992. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign.

de Moura, P. J. L. 2003. *Logtalk. Design of an Object-Oriented Logic Programming Language*. Ph.D. Dissertation, Universidade da Beira Interior, Departamento de Informatica, Covilha.

Jackson, P. 1999. *Introduction to Expert Systems*. Addison–Wesley, 3rd edition. ISBN 0-201-87686-8.

Ligęza, A.; Wojnicki, I.; and Nalepa, G. 2001. Tab-trees: a case tool for design of extended tabular systems. In et al., H. M., ed., *Database and Expert Systems Applications*, volume 2113 of *Lecture Notes in Computer Sciences*. Berlin: Springer-Verlag. 422–431.

Ligęza, A. 2006. *Logical Foundations for Rule-Based Systems*. Berlin, Heidelberg: Springer-Verlag.

Maurer, F., and Ruhe, G., eds. 2004. *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004*.

Mellor, S. J., and Balcer, M. 2002. *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Miller, J., and Mukerji, J. 2003. *MDA Guide Version 1.0.1*. OMG.

Nalepa, G. J., and Ligęza, A. 2005. Conceptual modelling and automated implementation of rule-based systems. In Krzysztof Zieliński, T. S., ed., *Software engineering : evolution and emerging technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, 330–340. Amsterdam: IOS Press.

Nalepa, G. J., and Zięcik, P. 2006. Integrated embedded prolog platform for rule-based control systems. In Napieralski, A., ed., *MIXDES 2006 : MIXed DESign of integrated circuits and systems : proceedings of the international conference : Gdynia, Poland 22–24 June 2006*, 716–721. Łodź: Technical University Lodz. Department of Microelectronics and Computer Science.

Nalepa, G. J. 2004. *Meta-Level Approach to Integrated Process of Design and Implementation of Rule-Based Systems*. Ph.D. Dissertation, AGH University of Science and Technology, AGH Institute of Automatics, Cracow, Poland.

Newell, A. 1982. The knowledge level. *Artificial Intelligence* 18(1):87–127.

Rash, J. L.; Hinchey, M. G.; Rouff, C. A.; Gracanin, D.; and Erickson, J. 2005. A tool for requirements-based programming. In *Integrated Design and Process Technology, IDPT-2005*. Society for Design and Process Science.

Ross, R. G. 2003. *Principles of the Business Rule Approach*. Addison-Wesley Professional, 1 edition.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2nd edition.

Sommerville, I. 2004. *Software Engineering*. International Computer Science. Pearson Education Limited, 7th edition.

Torsun, I. S. 1995. *Foundations of Intelligent Knowledge-Based Systems*. London, San Diego, New York, Boston, Sydney, Tokyo, Toronto: Academic Press.

von Halle, B. 2001. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. Wiley.