Knowledge-Based Approach to the Executable Design Concept *

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 describes the Executable Design Concept which is one of the main components of HeKatE: Hybrid Knowledge Engineering methodology. HeKatE project aims at developing a methodology and supporting technologies for Software Engineering based on Knowledge Engineering methods and paradigms. The Executable Design is a concept which allows to close the so-called *semantic gap* which exists between software design and its implementation. Thanks to the HeKatE ARD/XTT approach software can be designed in a new, declarative, knowledge-based way. The Executable Design assures that ARD/XTT design can be easily turned into a running application without any other efforts such as coding. The paper presents the Executable Design from the methodological and technological points of view of the HeKatE project.

1 Introduction

An effective development of complex software remains a challenge in software engineering. To cope with it, new design approaches are researched, and advanced programming languages are introduced. In order to build conceptual models of business logic, software engineers apply Artificial Intelligence (AI) methods.

Knowledge-based systems (KBS) are an important class of intelligent systems in the field of AI [1]. 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 [2,3]. Rule-based systems (RBS) constitute today one of the most important classes of KBS. However, building real-life KBS is a complex task. Since their architecture is fundamentally different from classic software, typical Software Engineering (SE) approaches cannot be applied efficiently. Some specific development methodologies, commonly referred to as *Knowledge Engineering* (KE), are required.

This paper presents a concept of a new software engineering approach based on knowledge engineering methods. In the paper some important features of both

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

KE and SE approaches are summarized in Sect. 2. Furthermore, common design problems encountered in the SE are outlined in Sect. 3. 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 in Sect. 4. The new SE approach is based on the concept of *executable design* (ED) discussed in Sect. 5. The HEKATE project aims at providing a platform for the ED, outlined in Sect. 6. The paper ends with concluding remarks in Sect. 7 where main features of HEKATE are summarized.

2 Knowledge to Software Engineering Approaches

In this section 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.

2.1 Principles of Knowledge Engineering

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* [4].

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 [3,5].

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

2.2 Principles of Software Engineering

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* [6]. 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 system 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.

3 Design Problems in Software Engineering

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.

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 [6].

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 the OMG [7]. 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 [8,9].

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.

3.1 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 [7]. 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 the design and its implementation [10].

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 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. 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* [11]. 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.

3.2 Problem statement

It could be summarized, that constant sources of errors in SE 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 stems from 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 of the methodologies, (mainly the *MDA*) and design approaches (mainly the *MVC* (Model-View-Controller) [12]) 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 KBS, thanks to the fact that the field is narrower and well formalized. The proof of concept is the Mirella [13] 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.

4 The HeKatE Project

The HeKatE project addresses the described problems. Basing on experiences with the Mirella project, it extends its RBS perspective towards SE.

Mirella proposed an integrated design process, that can be considered a *top-down hierarchical design methodology* It 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 (*eXtended Tabular Trees*) [13], a hybrid knowledge representation.

1. *Conceptual modeling*, in which system attributes and their functional relationships are identified; during this design phase the ARD (*Attribute-Relationship Diagrams*) [14,3] modelling method is used. It allows for specification of functional dependencies of system attributes using a visual representation. ARD 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. 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.

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. The main idea behind *XTT* [13] 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 (OAV) [3]. 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.
3. *Physical design*, in which a preliminary Prolog-based *implementation* is carried out. Using the predefined XTT translation it is possible to automatically build a prototype. It uses Prolog-based meta-language for representing XTT knowledge base and rule inference.

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.

A principal idea in the *HeKatE* 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 the use of a Prolog-based representation. The logical, Prolog-based core (the *logic core*) would be then embedded into a business application, or an 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* [15] (`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 Prolog. The logical design which specifies the knowledge base, which becomes an application, executable by a runtime environment, combining an inference engine and classic language runtime (e.g. Java Virtual Machine – JVM).

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.

5 The Executable Design Concept

The *executable design* concept (ED) aims at solving the main problems outlined at the previous section. 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 [10]. In order to do so the following elements should be developed: 1) a rich and expressive design method, 2) a high-level runtime environment, and 3) 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. Instead of developing a single robust platform such as Java, the focus should be on the possible integration of runtime platforms.[1]

The development of a 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 [16] 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, thus fulfilling a premise of the ED. However, it must be pointed out the the current state of the *xUML* is unclear, and its applications are limited.

A very important and influential concept concerns the so-called *design patterns* [17]. The idea is to identify certain *patterns* on the design level, and use them as the foundation for the future design. The design patterns are usually

[1] At the first sight this approach could look similar to the .NET platform, which aimes at providing a common runtime for different languages. However, *HeKatE* is different, because it aims at offering a different *knowledge-based design* paradigm.

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. What is more 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.

The application is knowledge-based, it is subject to modification both while being developed and executed. That is why the runtime environment for ED should be based on a dynamic language [18,19]. What is more, the application is just another representation of the designed knowledge base, expressed in a high level language, it is always possible to translate it back into the design this way.

The runtime environment is a compound of technologies constituting a virtual machine. The main component would be an interpreter of one of the languages mentioned earlier. Other components are to provide communication with environment: input/output, user interface, interaction with the operating system etc. The virtual machine technology proved to be a robust solution (JVM, .NET) and a theme of ongoing research (LLVM, `www.llvm.org`, HLVM, `www.hlvm.org`, Parrot, `ww.parrotcode.org`). Software, once designed, can be run on any hardware platform supporting the runtime environment. It is so-called *design-once-run-everywhere* which paraphrases *write-once-run-everywhere*: *write* is replaced by *design*, since there is no coding (implementing, writing) stage.

6 Executable Design in HeKatE

One of the main goals of *HeKatE* is to offer an effective platform for the ED.

In order to solve 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 than R2D2C (*Requirements To Design To Coding*) [11].

The *executable design* concept is presented in Fig. 1. It is based on the ARD/XTT mothodology. 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. Since ARD is hierarchical, it makes XTT-based knowledge hierarchical as well. Since knowledge base is hierarchical it can be managed easier than a monolithic one. An application model based on combined XTT 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.

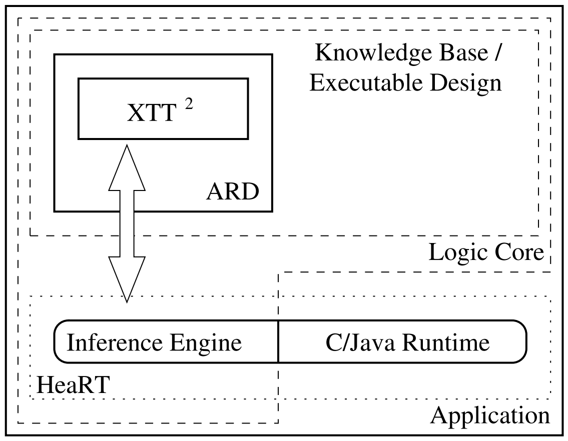

Figure 1. *Executable Design* Concept

From representing actual Knowledge to running application there is a clear transition. XTT is both knowledge representation and design methodology. However, there is a transformation needed for actually running XTT. XTT-based knowledge is translated into Prolog language based rules and executed by HeaRT. This translation is lossless: a reverse translation: from Prolog based representation into XTT is possible. Proper 1:1 translation is guaranteed. The Prolog based representation is not just another translation, it is the same knowledge base.

The HeKatE project provides means for the design and implementation of software logic, and the integration of this logic with the presentation layer. It allows 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 is 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. 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 design. 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. Also, 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 (platforms) 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.

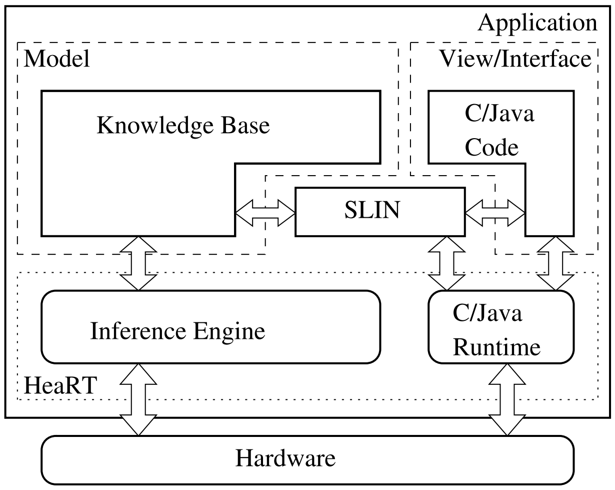

Figure 2. Multiparadigm Hybrid Approach: an Application

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 project 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*. Other software classes, such as general purpose or scientific software are also considered. Several well-documented test cases for HeKatE have been identifed so far, including the classic *UServ* example from the Business Rules Forum (see: `www.businessrulesforum.com/2005_Product_Derby.pdf`). When it comes to classic programming cases, some limitations of the approach can be shown, most important of them include complex data structures manipulation.

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 Analysis 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*.

7 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: ARD, XTT, knowledge representation methods and Prolog-based implementation.

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 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, while being a work in progress, the proposed methodology is going to provide an alternative for contemporary Software Engineering approaches.

References

1. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice-Hall (2003)
2. Jackson, P.: Introduction to Expert Systems. 3rd edn. Addison–Wesley (1999) ISBN 0-201-87686-8.
3. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
4. Newell, A.: The knowledge level. Artificial Intelligence **18**(1) (1982) 87–127
5. Torsun, I.S.: Foundations of Intelligent Knowledge-Based Systems. Academic Press, London, San Diego, New York, Boston, Sydney, Tokyo, Toronto (1995)
6. Sommerville, I.: Software Engineering. 7th edn. International Computer Science. Pearson Education Limited (2004)
7. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. OMG. (2003)
8. Ross, R.G.: Principles of the Business Rule Approach. 1 edn. Addison-Wesley Professional (2003)
9. von Halle, B.: Business Rules Applied: Building Better Systems Using the Business Rules Approach. Wiley (2001)
10. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
11. 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)
12. 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)
13. Nalepa, G.J.: Meta-Level Approach to Integrated Process of Design and Implementation of Rule-Based Systems. PhD thesis, AGH University of Science and Technology, AGH Institute of Automatics, Cracow, Poland (September 2004)
14. Nalepa, G.J., Ligęza, A.: 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., Amsterdam, IOS Press (2005) 330–340
15. de Moura, P.J.L.: Logtalk. Design of an Object-Oriented Logic Programming Language. PhD thesis, Universidade da Beira Interior, Departamento de Informatica, Covilha (2003)
16. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. 1st edn. Addison-Wesley Professional (2002)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. 1st edn. Addison-Wesley Pub Co. (1995)
18. Norvig, P.: Design patterns in dynamic programming. Tutorial at Object World, Boston, MA (May 1996) Tutorial slides at http://norvig.com/design-patterns/.
19. Sullivan, G.: Advanced programming language features for executable design patterns (2002)