Collaboration and the Knowledge Economy: Issues, Applications, Case Studies
Paul Cunningham and Miriam Cunningham (Eds)
IOS Press, 2008 Amsterdam
ISBN 978–1–58603–924-0

Comprehensive Developer Support for Rule-Based Programming

Valentin ZACHARIAS, Andreas ABECKER
FZI Forschungszentrum Informatik,
Haid-und-Neu-Str. 10-14, D-76131 Karlsruhe, Germany
Tel: +49 721 9654-802; Fax: +49 721 9654-803; Email: firstname.lastname@fzi.de

Abstract: Recent years saw a proliferated interest in rule formalisms for expressing business logic, compliance rules, consistency rules, etc. As the need for widespread adoption of rule languages arises and the user basis shall be extended from expert developers towards business experts, a better understanding of the real pros and cons of rule formalisms is necessary, and the creation of much more comfortable and powerful software engineering methods and tools fur rule-knowledge bases is indispensable. In this paper, we sketch the respective state of research and practice and present a comprehensive development tool suite for rule bases.

Keywords: knowledge technologies; rule-based programming; debugging; knowledge engineering

1. Introduction

Forward-chaining and backward-chaining rules have been discussed in Artificial Intelligence for about three decades and are still the predominant formalism for knowledge encoding in Expert Systems and Intelligent Advisor Systems. However, while Expert Systems never quitted the status of a niche application for high-end software, the recent years saw a renewed and enforced interest in rules (both logic-based and not logic-based), fuelled by two major trends in IT:

- On one hand, the growing interest in Semantic Web and Semantic Technologies (cp. [1]) where rule formalisms represent an extension of today's modelling capabilities in order to express richer conceptual knowledge on top of and as part of ontologies, knowledge bases and metadata.
- On the other hand, the growing adoption of business-rule approaches for building more modular and flexible, easier to maintain, business applications, e.g., in the compliance area, which shall also empower business experts to encode their domain knowledge in a software application (cp. [2]).

Although there are few people working on specific topics of rule-base engineering, e.g., visualization of rule bases, it is nevertheless obvious that there is surprisingly few work in areas such as modern methodologies for rule-base engineering, rule-base debugging, and rule-based programming environments in general – all that areas which would be necessary to guarantee that rule-based programming can be done easily and efficiently in a quality-assured manner, also in larger software-development contexts.

Hence it is the aim of our work which is cursorily presented in this paper, to contribute to a modern theory and practice of rule-base engineering – based on a thorough understanding of rule-base engineering in practice, equipped with a reasonable methodological fundament and supported by a comprehensive set of tools.

Our target audience are primarily (applied) computer scientists, software or knowledge engineers dealing with rules in the Expert System, the Business Rules, or the Semantic Web

area, who are interested in the state of practice and possible steps forward regarding user-friendly rule-base engineering.

2. Objectives

The objectives of our work were:

1. to understand what the problems of large-scale, real-world, rule-base engineering are; and to identify common misconceptions and misunderstandings in this respect
2. to identify the relevant prior work in several computer science sub-disciplines (software engineering, expert systems, semantic web, ..) which might contribute to a modern theory and practice of rule-base engineering
3. to develop a method for modern rule-base engineering
4. to deploy an integrated set of tools for comprehensively supporting this method

Our answers to these four challenges will at least be sketched in the following. References to more comprehensive presentations are included.

3. Methodology

Regarding the *research methodology* followed in this work, we can name several kinds of academic and empirical evidence our developments were based on:

- on action-research of one of the authors who participated for more than a year in several projects of one of Germany's leading commercial providers of rule-based systems
- on a questionnaire-based survey of the current status of rule-based development and its problems in business and academia (with a return of ca. 50 rule-base developers who answered our questions)
- on the study of the relevant scientific literature in software engineering, expert system development, verification and validation of forward-chaining rule bases, declarative programming and debugging support, starting with the late 1980ies

Based on our action-research experience, we identified a number of reasons for the difficulties of rule-base debugging, like, for instance (cp. [4]):

- **The One Rule Fallacy:** It is often thought that rule-base debugging is easy because knowledge is localized, side-effects are minimized and, thus, the single rule is the only and most important object for debugging. Practice shows that the automatic recombination of rules at run-time in ways not envisioned at build-time, necessitates testing of rule interaction on as many as possible diverse problems.
  - This led to the idea of transferring test-coverage metrics from software engineering in procedural programming, to the area of rule programming.
- **The Problem of Opacity:** Failed interactions between rules are the most important source of errors during rule-base creation. At the same time, as this interaction shall be opaque to the user at run-time, it is normally not visualized in development tools.
- **The Problem of Interconnection:** Rule interactions are managed by the inference engine, so everything is potentially relevant to everything else. This complicates fault localization, because bugs appear in seemingly unrelated parts of the rule base. Another consequence is that even a single fault can cause a large portion of (or even all) test cases to fail.
  - These two phenomena motivated our work on rule-base visualization for getting a better overview of rule effects and interrelationships.
- **The Problem of Error Reporting:** One of the most common symptoms of an error in a rule base is a query that (unexpectedly) does not return any result. In such a case most inference engines give no further information that could aid the user in error localization.

- This inspired our work in automated debugging, starting from work on explanation generation in Expert Systems.

The practical work clearly showed that (i) there is much less refined development support for rule programming, compared to imperative programming, and (ii) that even most available tools for rule-base debugging were pretty much aligned with the procedural debugging paradigm which is apparently not perfectly suited for declarative programming.

In order to base our work on a broader empirical basis than only our own individual experience, we performed a survey among 50 rule-base programmers in early 2008. Some indicative, exemplary results are sketched below.

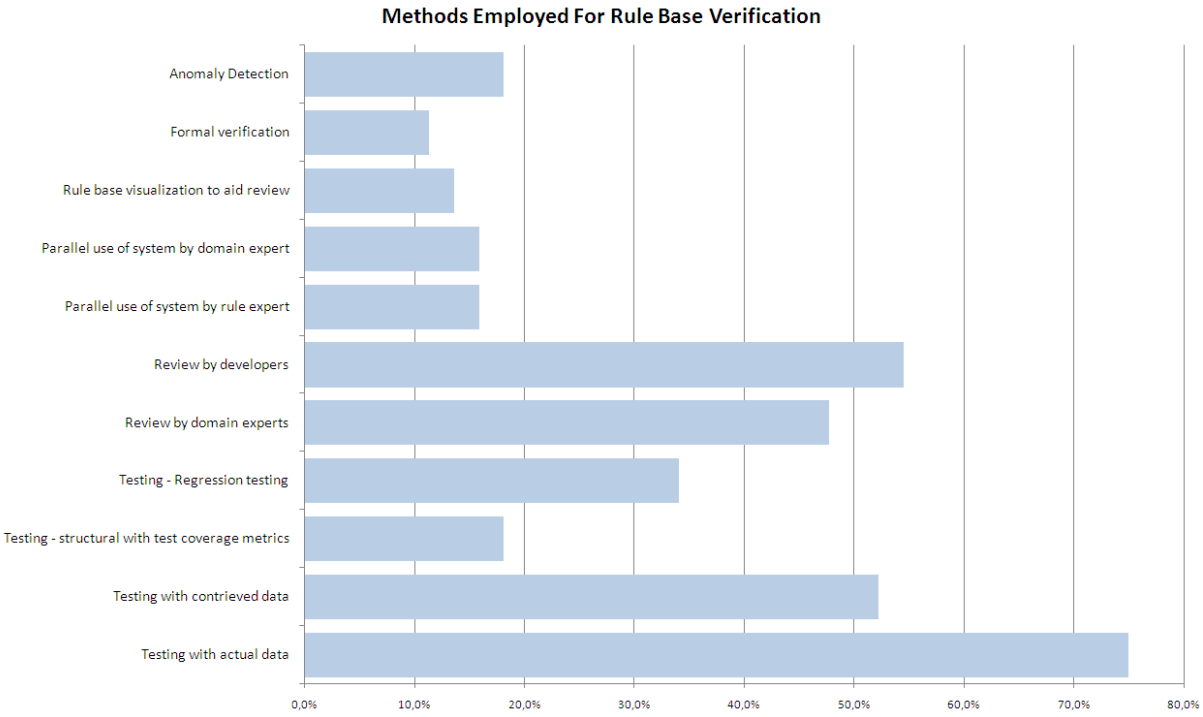

Figure 1: Survey results regarding rule-base verification methods in practical use

As Figure 1 suggests (we show the percentage of developers who said they employ a certain rule-base verification method), the majority of rule-base developers works with relatively unspecific methods known from imperative programming (like testing with actual data, software reviews, …) while modern, systematic approaches (like testing with coverage metrics) are less popular, and specific methods geared towards the supposed strengths and particularities of declarative programming (like anomaly detection, formal verification, or even parallel use in rapid prototyping cycles) are the least used ones.

Figure 2 illustrates a similar insight regarding the debugging tools in use: The most prominent tool is clearly procedural debugging – more than 50% of rule-base development projects use either a graphical or a textual procedural debugger. Procedural debuggers are followed by explanations, still used by almost a quarter of the projects. All other approaches are only used very seldom and are usually not available in industrial-strength implementations.

Altogether, the survey corroborated the impressions we had already from our personal experience: rule-base development in practice does not specifically take into account the particular characteristics of declarative and rule-based programming; practitioners as well as researchers work mainly with tools not originally built for that purpose and usage context; much prior work in potentially contributing fields is not fully exploited or is even completely ignored; the majority of practitioners seems not to follow specific knowledge engineering methodologies especially geared towards rule-base engineering.

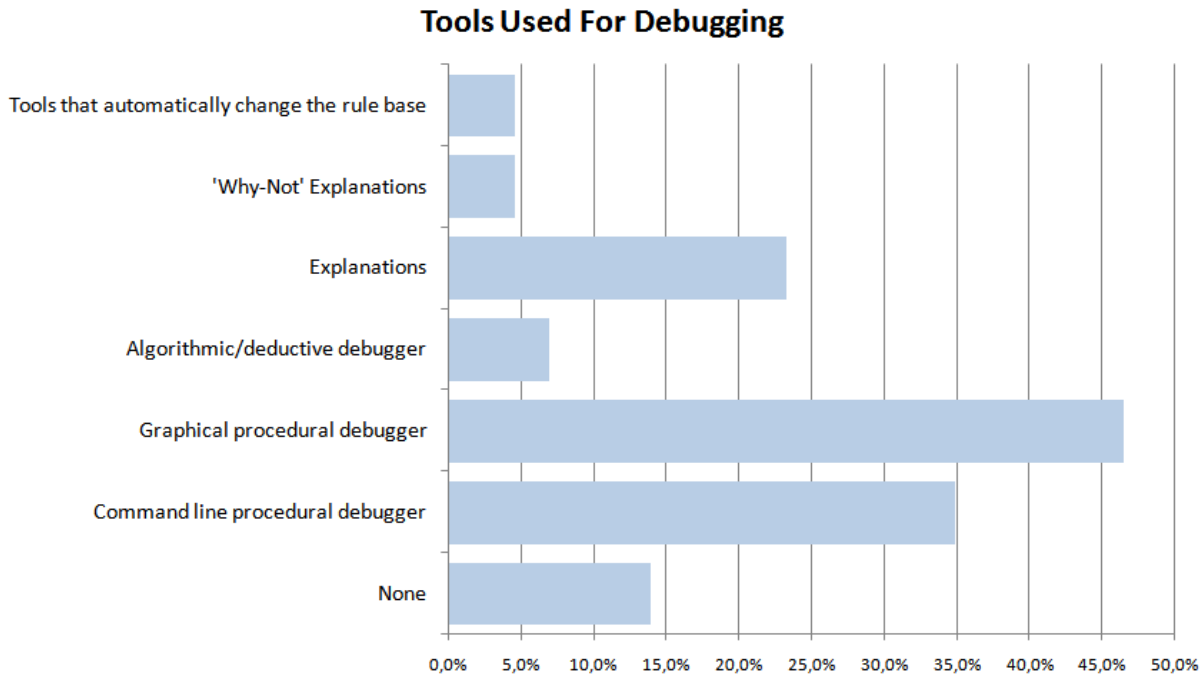

Figure 2: Survey results regarding rule-base debugging tools in practical use

In order to overcome such deficiencies and achieve a higher level of efficiency, we designed an agile development process for rule-base engineering, based on some principles derived from Extreme Programming (XP). The method is grounded in some main ideas:

(1) *Program first, knowledge second:* in contrast to traditional views on declarative approaches, we consider rule-base development still mostly as a task-driven programming activity.

(2) *Interactive rule creation:* instantly trying out partial solutions at any time in the development phase, must be supported.

(3) *Explanation is documentation:* as explanation plays a predominant role for debugging atbuild-time and user acceptance at run-time, explanation and documentation is treated as one activity.

The method, based on these ideas and inspired by well-known process templates from XP, is explained in more detail in [5].

Based on the considerations and achievements described so far, thinking about comprehensive tool support for rule-base engineering leads to the idea of an integrated set of tools, the individual tools derived from manifold prior work in software engineering and knowledge engineering, synergetically working together in an iterative rule-base engineering process according to the principles sketched above. Figure 3 below illustrates the intended interaction of several supporting functionalities. Most importantly, the figure shows that test, debug and program activities should be fully intertwined and interconnected – as mandated by the interactivity principle described above. These activities are supported by test coverage, anomaly detection and visualization as major support tools - which explained very shortly in the following section.

4. Technology description

Because (a) “classic” work in forward-chaining systems as well as today’s business rules, (b) the state of the art in logic programming, and (c) several current Semantic Web rules (like JENA rules or SWRL) are all based on different rule languages, with different syntax and semantics, our first step before realizing our tool suite, was to express the most relevant languages in one common form, namely Normal Logic Programs. This allowed to transfer

approaches and results from one field to the others and allowed to integrate contributions from different fields into one unifying framework. This transformation is straightforward, yet cumbersome – and, not yet done elsewhere (at least to our knowledge). Details will be presented in the first author's dissertation to be finished in late 2008.

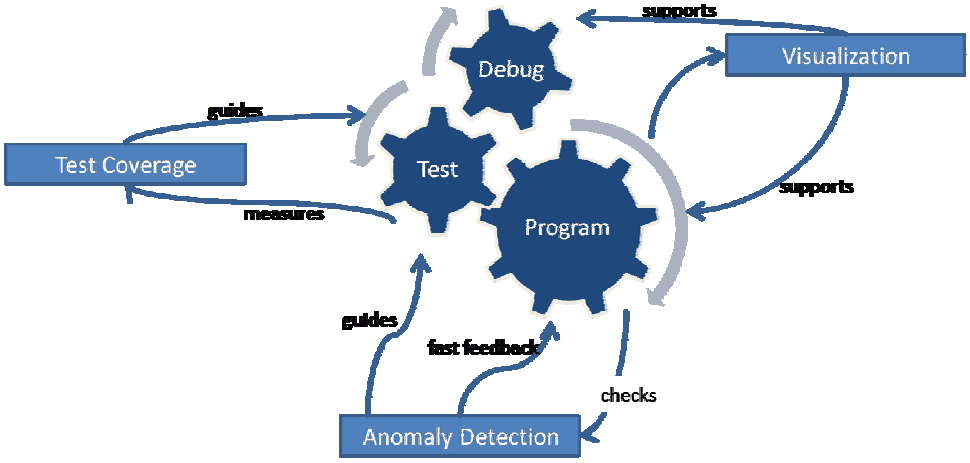

Figure 3: Tool functionalities playing together in iterative rule-base engineering

The following subsections sketch the individual software modules developed. For one module, the explorative debugging tool, we go into some more detail. The other modues are described more elaborately, e.g., in [3; 4; 6]. Further methods investigated comprise algorithmic debugging [10; 11] as well as methods based on explanation generation in Expert Systems. After having sketched the three modules indicated in the Figure above, we explain in some more detail the idea of **explorative debugging** which is a specific, new, outcome of this thread of work.

4.1 Anomaly detection

Anomaly detection is a well-known static verification approach extensively discussed in verification and validation of forward-chaining rule systems for heuristically identifying and showing symptoms of possible errors in a rule-knowledge base [7]. Anomalies are often concerned with the hidden interaction between rules and are commonly subdivided into four large categories: (i) Circularity in rule sets; (ii) Ambivalence, especially contradictory rules; (iii) Deficiency, like input data never used or other symptoms of missing parts in the rule base; and (iv) Redundancy in the rule base. Detected anomalies are reported immediately to the developer; anomalies that the developer does not understand as an error, can form a guidance for parts of the rule base that should be thoroughly tested.

4.2 Visualization

The identification and visualization of rule-base structure and rule interaction has been discussed in [8]:

- At build-time, the **static structure** is based solely on the rules (and not the facts) and tries to identify the interactions between rules that could happen, if the rule base was to be used with the right facts.
- At run-time or test-time, the **dynamic structure** of a rule base is created from the actual rule interactions that happen during query evaluation.

4.3 Test management

Tools to facilitate systematic testing of rule bases include test-case editors and are based on test coverage metrics for rules bases. In [9], several coverage metrics for F-Logic have been defined and implemented. In the same work, the metrics-based approach has been

complemented by a graphical approach that allows the user to assess the completeness of a set of test cases by a visualization.

4.4 Explorative debugging

Explorative Debugging works on the declarative semantics of a rule base and lets the user navigate and explore the inference process. It enables the user to use all available knowledge to quickly find problems and to learn about the program at the same time. An Explorative Debugger is not only a tool to identify the cause of occurring problems, but also a tool to try out a program and learn about its inner working.

Explorative Debugging puts the focus on rules. It enables the user to check which inferences a rule enables, how it interacts with other parts of the rule base and what role the different rule parts play. Unlike in procedural debuggers, the debugging process is not determined by the procedural nature of the inference engine, but by the user who can use the logical/semantic relations between the rules to navigate. Particular functionalities of an explorative debugger comprise:

- Show the inferences that a rule enables.
- Visualize the logical/semantic connections between rules and show how rules work together to arrive at a result. It supports the navigation along these connections.
- Enable further exploration of the inference by digging down into the rule parts.

An explorative debugger is integrated into the development environment and can be started quickly to try out a rule as it is formulated. The screenshot in Figure 4 below shows the explorative debugger from the Trie! project[i]. The main elements of the user interface are:

- At the top, the **Rule Details** show the current rule and its comment. UI elements allow to explore, how the rule would behave if some conditions would be removed.
- In the middle, the **Rule Firings** show the inferences that are enabled by the rule shown. In the example, it is shown that the rule can conclude that JohnsUncle is of type father, based on A=JohnsUncle and B=John.
- In the bottom right, a **Proof tree** is shown for the result selected in the rule firings part. The user can click on any of the rules in the proof tree to navigate there.
- On the bottom left, the **Depends On** part shows relations to other rules. In the example, only one rule is shown that is contained in the proof tree. In general, this part also shows rules that are not involved in current firings of the rule, but that are in other ways related to the rule – for instance, rules that could deduce facts that the current rule could use, or rules that - with only a small change - could be relevant for the current rule. The user can click on any rules shown in this part to navigate there.
- **Navigation Elements** on the top right, allow to jump to any rule, jump back to the query, customize the way in which rules are displayed and allow to go forward or backwards analog to similar buttons in web-browsers.

Explorative Debugging is a paradigm that keeps the user in control of the debugging process and at the same time stays at the declarative level, freeing the user from having to worry about the details of the inference engine. Explorative Debugging builds on the ideas from Explanation systems but is not limited to queries that have a result; it can be easily combined with the automatic why-not explanations.

As an example, consider the following rule base, seeded with an example error and with an additional fact: `male(Mike).`

```
motherOf(A,X) <-  female(A), child(A,X)  # Rule motherOf
fatherOf(A,X) <-  male(A), child(X,A)    # Rule fatherOf;
                                            error, should be child(A,X)
parent(Y)     <- fatherOf(Y,X)           # Rule parent_1
parent(Y)     <- motherOf(Y,X)           # Rule parent_2
male(Peter)
child(Peter, Mike)
```

For the query `<- parent(Y)`
a debugging session with an explorative debugger might look as follows:

- The debugging session starts with the user looking at the query. No result is shown for the query, but the rules parent_1 and parent_2 will appear in the depends-on part. The user starts with parent_1 and clicks on it to investigate it further.
- The debugger then displays parent_1. Again the Rule Firings part is empty, but the rule fatherOf appears in the depends-on part. The user clicks on it.
- The rule fatherOf is displayed, again no Rule Firings are shown, but the colour of the two conditions (male(A) and child(X,A)) will indicate that each of the two conditions is satisfiable. The user then selects to only look at the results for the child(X,A) condition and thereby finds the error.

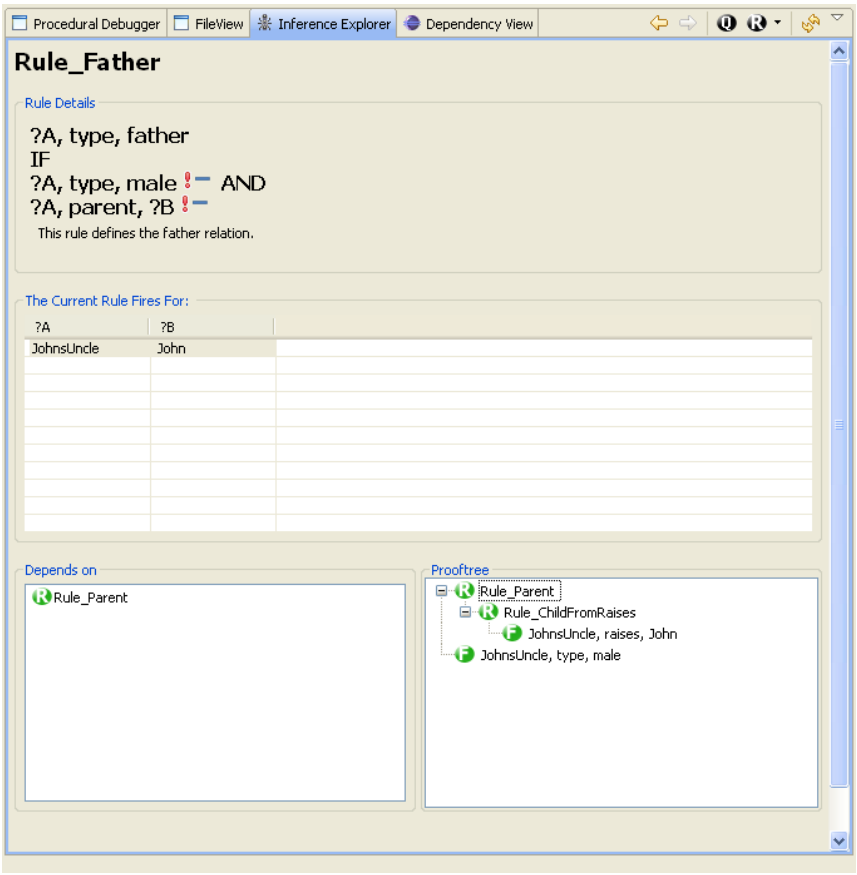

Figure 4: Explorative debugger of the Trie! project

5. Conclusions

In the near future, it can be expected that rule-based programming may be required from many domain specialists who are non-expert rule language users. In this case, user-friendly and powerful development suites will have an utmost importance. Just now, companies like SAP and IBM try to or really acquire small and medium sized companies with a profile in rule management which is a clear indicator that business promises are seen there.

Surprisingly, there is extremely few work in the rule engineering area, at this moment, which can be seen, e.g., from the very few, partially relatively old, related works cited in this paper. There is clearly room for more research and development in this topic. Hence we hope that our comprehensive, practice-driven and empirically validated, approach – which combines methodological as well as tool support and which integrates useful contributions from manifold, also older, research areas – will promote progress in this direction. An

overall lesson learned may be that research in this direction must be more empirically oriented and better settled in real-world projects than usual in much academic research.

All the methods described in this paper have been prototypically implemented and tested – however, for different rule languages; the unifying framework for Normal Logic Programs is still under work; more extensive lab experiments and/ or longitudinal field studies for assessing the practical usefulness, are still missing. Clearly, a realistic and promising mid-term goal is to include the functionalities as described here, in the commercially used rule-base engineering suites available.

Acknowledgment

The work presented in this paper has partially been supported by the European Commission in the project "NEPOMUK – The Social Semantic Desktop" (grant no. FP6-027705) and by the German National Ministry for Education and Research (bmb+f) in the Project "Im Wissensnetz – Vernetzte Informationsprozesse in Forschungsverbünden" (grant no. 01C5979 - 01C5982),

References

[1] M. Kifer, J. de Bruijn, H. Boley and D. Fensel (2005): A Realistic Architecture for the Semantic Web. In: *Proc. RuleML-2005*.

[2] K. Seer (2005): The 2005 Business Rules Awareness Survey. *Business Rules Journal* 6(8).

[3] V. Zacharias, A. Abecker (2007): On Modern Debugging For Rule-Based Systems. In: *19th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE-2007).*

[4] V. Zacharias (2008): Rules as a Simple Way to Model Knowledge – Closing the Gap between Promise and Reality. In: *10th Int. Conf. on Enterprise Information Systems (ICEIS-2008).*

[5] V. Zacharias (2007): The Agile Development of Rule Bases. In: *16th Int. Conf. on Information Systems Development (ISD2007).*

[6] V. Zacharias (2008): *The Debugging of Rule Bases.* Submitted for publication.

[7] A. Preece, R. Shinghal (1994): Foundation and Application of Knowledge Base Verification. *Int. Journal of Intelligent Systems* 9(8):683-701.

[8] V. Zacharias (2007): Visualization of Rule Bases – The Overall Structure. In: *7th Int. Conf. on Knowledge Management (I-Know 2007)*, Special Track on Knowledge Visualization and Knowledge Discovery.

[9] F. Kleiner (2006): *Testvollständigkeit von F-logik Wissensbasen.* Diploma Thesis. Fakultät für Informatik. Universität Karlsruhe, Germany. *In German.*

[10] M. Ducasse, J. Noy (1994): Logic Programming Environments: Dynamic Program Analysis and Debugging. *Journal of Logic Programming* 19(20):351-384.

[11] M. Stumptner, F. Wotawa (1998): A Survey of Intelligent Debugging. *AI Communications* 11(1):35-51.

[i] http://code.google.com/p/trie-rules/