RULES AS SIMPLE WAY TO MODEL KNOWLEDGE
Closing the Gap between Promise and Reality

Valentin Zacharias
FZI, Haid-Und-Neu Strasse 10-14, Karlsruhe, Germany

Keywords: Rule based systems, F-logic, knowledge acquisition, rule based modeling, knowledge modeling, verification, anomaly detection.

Abstract: There is a considerable gap between the potential of rules bases to be a simpler way to formulate high level knowledge and the reality of tiresome and error prone rule bases creation processes.
Based on the experience from three rule base creation projects this paper identifies reasons for this gap between promise and reality and proposes steps that can be taken to close it. An architecture for a complete support of rule base development is presented.

1 INTRODUCTION

Since their conception rule languages have been heralded as a simpler and more natural way to build computer systems, compared to both imperative programming and other logic formalisms. This idea rests on three observations:

- Rule languages are (mostly) declarative languages that free the developer from worrying about *how* something is computed. This should decrease the complexity for the developer.
- The If-Then structure of rules resembles the way humans naturally communicate a large part of knowledge.
- The basic structure of a rule is very simple and easy to understand.

However, the authors have observed the creation of rule bases in three different projects and found this promise of simplicity to be elusive. If anything the creation of rule bases was more error prone, more tiresome and more frustrating than the development with modern imperative languages.

This paper is an attempt to reconcile the promise of simplicity with reality. It identifies why rule bases are not currently simple to build and proposes steps to address this. This paper's structure is as follows: section two gives a short overview of the three projects that form the input for this analysis. Section three discusses what kinds of problems were encountered. The next three sections then discuss overall principles to better support the creation of rule bases, an architecture of tool support for the development process and finally debugging as a particularly important problem. The paper discusses two commonly voiced counter arguments against ideas presented here and then concludes.

2 EXPERIENCES

The analysis presented in this paper is based on the experience from three projects.

- *Project Halo*[1] is a multistage project to develop systems and methods that enable domain experts to model scientific knowledge. As part of the second phase of Project Halo six domain experts were employed for 6 weeks each to create rule bases representing domain knowledge in physics, chemistry and biology.
- *Online Forecast* was a project to explore the potential of knowledge based systems with respect to maintainability and understandability. Towards this end an existing reporting application in a human resource department was re-created as rule based system. This project was performed by a junior developer who had little prior experience with rule based system. Approximately 5 person months went into this application.
- The goal of the project *F-verify* was to create a rule base as support for verification activi-

[1]http://www.projecthalo.com

ties. It models (mostly heuristic) knowledge about anomalies in rule bases. It consists of anomaly detection rules that work on the reified version of a rule base. This project was done by a developer with experience in creating rule bases and took 3 month to complete.

All of these projects used F-logic (Kifer et al., 1995) as representation language and the Ontobroker inference engine (Decker et al., 1999). The editing tools differed between the projects: Project Halo used the high level, graphical tools developed in the project. Online Forecast used a simple version of Ontoprise's[2] OntoStudio. F-verify used Ontostudio together with prototypical debugging and editing tools.

The use of only one rule formalism obviously restricts the generality of any statements that can be made. However, F-logic is very similar to some of the rule languages under discussion for the Semantic Web (Kifer et al., 2005) and it is based on normal logic programs - probably the prototypical rule language. The authors examined literature and tools to ensure that tool support identified as missing is indeed missing from rule based systems and not only from systems supporting F-logic.

Methods used for the creation of the rule bases differed between the three projects:

- Project Halo used the document driven knowledge formulation process: scientific textbooks were used as informal models of domain knowledge. Using the textbooks as guideline and to structure the process, domain experts created the knowledge base. Graphic, high level tools were employed to allow the domain experts to concentrate on modeling knowledge without having to worry about the technicalities of implementation.
- Online Forecast began with the creation of informal models of the domain knowledge as diagrams and textual descriptions. These models were created in close cooperation with domain experts. In a second step these informal models were implemented. Results from test cases were again cleared with the experts and the knowledge base was refined until it matched their expectations.
- F-Verify used an informal iterative process. New heuristics would be defined in natural language and then be directly implemented. The domain model (the reified rules) existed already and was reused.

All of these projects have been created using relatively lightweight methods that focus more on the actual implementation of rules than on high level models - as is to be expected for such relatively small projects, in particular when performed by domain experts or junior programmers. The analysis, methods and tools presented in this paper are geared towards this kind of small, domain expert/end user programmer driven projects.

[2]http://www.ontoprise.de

3 ANALYSIS

In the three projects sketched above the authors found that the creation of working rule bases was an error prone and tiresome process. In all cases the developers complained that creating correct rule bases takes too long and in particular that debugging takes up too much time. Developers with prior experience in imperative programming languages said that developing rule bases was more difficult then developing imperative applications. While part of these observations can be explained by longer training with imperative programming, it still stands in a marked contrast to the often repeated assertion that creating rule bases is simpler. We identified six reasons for this observed discrepancy.

- **The One Rule Fallacy.** Because *one* rule is relatively simple and because the interaction between rules is handled automatically by the inference engine, it is often *assumed* that a rule base as a whole is automatically simple to create. However, the inference engine obviously combines the rules only based on how the user has specified the rules; and it is here - in the creation of rules in a way that they can work together - that most errors get made. Examples for errors affecting the interaction of rules are the use of different attributes to represent the same thing or two rules being based on incompatible notions of a concept.
  Hence a part of the gap between the expected simplicity of rule base creation and the reality can be explained by naive assumptions about rule base creation. Rule based systems hold the promise to allow the automatic recombination of rules to tackle problems not directly envisioned during the creation of the rule base. However, it is an illusion to assume that rules created in isolation will work together automatically in all situations. Rules have to be tested in their interaction with other rules on as many diverse problems as possible to have a chance for them to work in novel situations.
- **The Problem of Opacity.** The authors experience shows that failed interactions between rules are the most important source of errors during rule base creation. At the same time it is this interaction between rules that is commonly not shown;

that is opaque to the user. The only common way to explicitly show the connections between rules is in a prooftree of a successful query to the rule base. This is unlike imperative programming, were the relations between the entities and the overall structure of the program are explicitly created by the developer, shown in the source code and the subject of visualizations.

- **The Problem of Interconnection.** Because rule interactions are managed by the inference engine, everything is potentially relevant to everything else[3]. This complicates error localization for the user, because bugs appear in seemingly unrelated parts of the rule base. Another consequence is that even a single error can cause a large portion of (or even all) test cases to fail[4]. This too was a common occurrence, in particular in Project Halo.
- **The No-result Case and the Problem of Error Reporting.** By far the most common symptom of an error in the rule base was a query that (unexpectedly) did not return any result - the no-result case. In such a case most inference engines give no further information that could aid the user in error localization. This is unlike many imperative languages that often produce a partial output and a stack trace. Both imperative and rule based systems sometimes show bugs by behaving erratically, but only rule based systems show the overwhelming majority of bugs by terminating without giving any help on error localization.
- **The Problem of Procedural Debugging.** All deployed debugging tools for rule based programs known to the authors are based on the procedural or imperative debugging paradigm. This debugging paradigm is well known from the world of imperative programming and characterized by the concepts of breakpoints and stepping. A procedural debugger offers a way to indicate a rule/rule part where the program execution is to stop and has to wait for commands from the users. The user can then look at the current state of the program and give commands to execute a number of the successive instructions. However a rule base does not define an order of execution - hence the order of debugging is based on the evaluation strategy of the inference engine. The execution steps of a procedural debugger are then for instance: the inference engine tries to prove a goal A or it tries to find more results for another goal B.
  Procedural Debuggers force the developer to learn about the inner structure of the inference engine. This stands in marked contrast to the idea that rule bases free the developer from worrying about *how* something is computed. The development of rule based systems cannot take full advantage of the declarative nature of rules, when debugging is done on the procedural nature of the inference engine.
- **Less Refined Tools Support.** Compared to tools available as support for the development with imperative languages, those for the development of rule bases often lack refinement. This discrepancy is a direct consequence from the fact that in recent years the percentage of applications built with imperative programming languages was much larger than those built with rule languages.

[3]At least in the absence of robust, user managed modularization

[4]For example by making a rule base unstratifiable

4 CORE PRINCIPLES

The authors have identified four core principles to guide the building of better tool support for the creation of rule bases. These principles where conceived either by generalizing from tools that worked well or as direct antidote to problems encountered repeatedly.

- **Interactivity.** To create tools in a way that they give feedback at the earliest moment possible. To support an incremental, try-and-error process of rule base creation by allowing trying out a rule as it is created.
  Tools that embodied interactivity proved to be very popular and successful in the three projects under discussion. Tools such as fast graphical editors for test queries, text editors that automatically load their data into the inference engine or simple schema based verification during rule formulation where the most successful tools employed. In cases where quick feedback during knowledge formulation could not be given[5], this was reflected immediately in erroneous rules and unmotivated developers.
  Interactivity is known to be an important success factor for development tools, in particular for those geared towards end user programmers (Ruthruff and Burnett, 2005; Ruthruff et al., 2004). Interactivity as a principle addresses many of the problems identified in the previous section by supporting faster learning during knowledge formulation. Immediate feedback after small changes also helps to deal with the problem of interconnection and the problem of error reporting.

[5]This was mostly due to very long reasoning times or due to technical problems with the inference engine

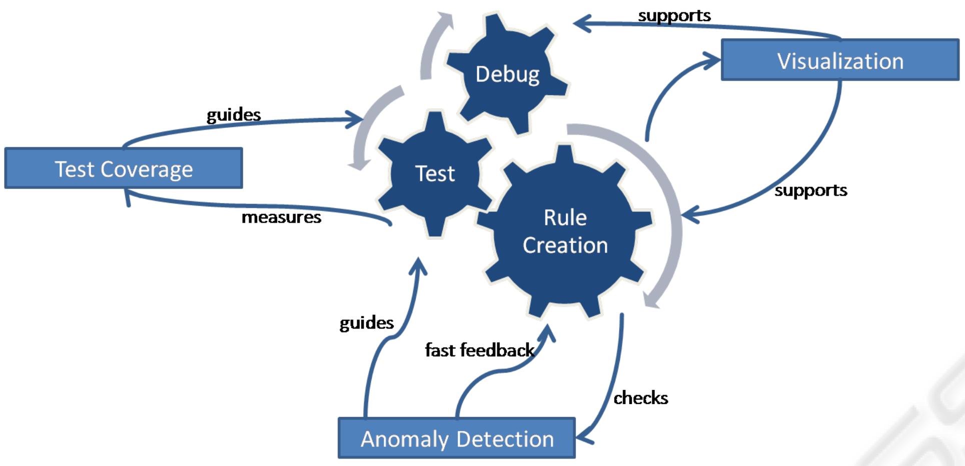

Figure 1: The overall architecture of tool support for the rule base development.

- **Visibility.** To show the hidden structure of (potential) rule interactions at every opportunity. Visibility is included as a direct counteragent to the problems of opacity and interconnection.
- **Declarativity.** To create tools in a way that the user never has to worry about the *how*; about the procedural nature of the computation. Declaritivity of all development tools is a prerequisite to realize the potential of reduced complexity offered by declarative programming languages. The declarititivity principle is a direct response to the problem of procedural debugging described in the previous section.
- **Modularization.** To support the structuring of a rule base in modules in order to give the user the possibility to isolate parts of the rule base and prevent unintended rule interactions. This principle is a direct consequence of the problem of interconnection. However, modularization of rule bases has been extensively researched and implemented (e.g. (Jacob and Froscher, 1990; Baroff et al., 1987; Grossner et al., 1994; Mehrotra, 1991) and will not be further discussed in this paper .

5 SUPPORTING RULE BASE DEVELOPMENT

The principles identified in the previous section need to be embodied in concrete tools and development processes in order to be effective. Based on our experience from the three projects described earlier and on best practices as described in literature (eg (Preece et al., 1997; Gupta, 1993; Tsai et al., 1999)) we have identified (and largely implemented) an architecture of tool support for the development of rule bases.

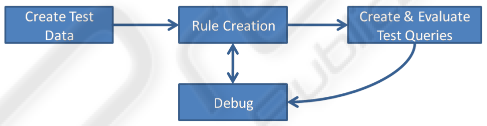

Figure 2: Test, debug and rule creation as integrated activities.

A high level view of this architecture is shown in figure 1. At its core it shows test, debug and rule creation as an integrated activity - as mandated by the interactive principle. These activities are supported by test coverage, anomaly detection and visualization. Each of the main parts will be described in more detail in the following paragraphs.

5.1 Test, Debug and Rule Creation as Integrated Activity

In order to truly support the interactivity principle the test, debug and rule creation activities need to be closely integrated. This stands in contrast to the usual sequence of: create program part, create test and debug if necessary. An overview of this integrated activity is shown in figure 2.

Testing has been broken up into two separate activities: the creation/identification of test data and the creation and evaluation of test queries. This has been done because test data can inform the rule creation process - even in the absence of actual test queries. Based on the test data an editor can give feedback on the inferences made possible by a rule while it is created. An editor can automatically display that (given the state of the rule that is being created, the contents of the rule base and the test data) the current rule allows to infer such and such. This allows the developer to get instant feedback on whether her intuition of what a rule is supposed to infer matches reality. When the inferences a rule enables do not match the expectations of the user she must be able to directly switch to the debugger to investigate this. In this way testing and debugging become integrated, even without a test query being present.

Hence we speak of integrated test, debug and rule creation because unlike in traditional development:

- test data is used throughout editing to give immediate feedback
- debugging is permanently available to support rule creation and editing, even without an actual test query. The developer can debug rules in the absence of test queries.

5.2 Anomalies

Anomalies are symptoms of probably errors in the knowledge base (Preece and Shinghal, 1994). Anomaly detection is a well established and often used static verification technique for the development of rule bases. Its goal is to identify errors early that would be expensive to diagnose later. Anomalies give feedback to the rule creation process by pointing to errors and can guide the user to perform extra tests on parts of the rule base that seem problematic. Anomaly detection heuristics focus on errors across rules that would otherwise be very hard to detects. Examples for anomalies are rules that use concepts that are not in the ontology or rules deducing relations that are never used anywhere.

Of the problems identified in section 3, anomalies detection partly addresses the problem of opacity, interconnection and error reporting by finding some of the errors related to rule interactions based on static analysis.

We deployed anomaly detection heuristics within Project Halo, both as very simple heuristics integrated into the rule editor and more complex heuristics as a separate tool. The anomaly heuristics integrated into the editor performed well and were well received by the developers, the more complex heuristics, however, took a long time to be calculated, thereby violated the interactivity paradigm, and weren't accepted by the users.

5.3 Visualization

The visualization of rule bases here means the use of graphic design techniques to display the overall structure of the entire rule base, independent of the answer to any particular query. The goals for such visualizations are the same as for UML class diagrams and other overview representations of programs and models: to aid teaching, development, debugging, validation and maintenance by facilitating a better understanding of a computer program/model by the developers. To, for example, show which other parts are affected by a change. The overview visualization of the entire rule base is a response to the problem of opacity.

Not being able to get an overview of the entire rule base and being lost in the navigation of the rule base was a frequent complaint in the three projects described in the beginning. To address this we created a scalable overview representation of the static and dynamic structure of a rule base, detailed descriptions can be found in (Zacharias, 2007).

6 DEBUGGING

In section 2 the current procedural debugging support was identified as unsuited to support the simple creation of rule bases. It was established that debuggers needs to be declarative in order to realize the full potential of declarative programming languages.

6.1 Previous Non-procedural Debuggers

The problem of debugging declarative systems has been investigated in numerous research projects in the past; two big threads of research can be identified:

- Algorithmic Debugging[6] (Shapiro, 1982; Stumptner and Wotawa, 1998; Naish, 1992; Brna et al., 1991; Pereira, 1986; Dershowitz and Lee, 1987) ; optimized algorithms that search an abstract representation of a programs execution for the part that is causing a bug, using either the user or a specification as oracle.
- Automatic Debugging[7] (Craw and Boswell, 2000; Craw and Sleeman, 1996; Ginsberg, 1988; Wilkins, 1990; Waterman, 1968; Ourston and Mooney, 1990; Richards, 1991; Wogulis and Pazzani, 1993; Raedt, 1992; Saitta et al., 1993; Morik and Emde, 1993), approaches that try to automatically identify (and possible correct) the error causing a bug, for instance by identifying the smallest

[6]Similar works have also been published under the terms Declarative Debugging, Declarative Diagnosis, Guided Debugging, Rational Debugging and Deductive Debugging

[7]The authors use automatic debugging as very broad broad combination of the areas of Automatic Debugging, Why-Not Explanation, Knowledge Refinement, Automatic Theory Revision and Abductive Reasoning

change to the rule base that would change the output to the expected value.

We reject both approaches as unsuited to be the *exclusive* debugging support within the architecture described here for the following reasons:

- Both approaches can be described as *computer controlled debugging*. Because the computer controls the debugging process, these tools can only use the information encoded in the rule base, not the domain knowledge and intuition of the developer. These approaches also do not empower the developer to learn more about the program. Many approaches just propose a change without giving the developer a chance to test and correct her expectations.
- These approaches are hard to reconcile with the interactivity principle and the integrated rule creation-test-debug development process because they depend on the existence of a test case, mostly together with the expected result.
- To the author's best knowledge none of the automatic approaches has ever proved a sufficient accuracy in identifying errors to find widespread adoption. Also many of these approaches rely heavily on the competent programmers hypotheses (DeMillo et al., 1978), that does not hold in all circumstances. This is particularly critical when no fallback, non-automatic debugging support is available to support the user in the misidentified cases.

6.2 Explorative Debugging

To address this shortcoming we have developed and implemented the Explorative Debugging paradigm for rule-based systems (Zacharias and Abecker, 2007). Explorative Debugging works on the declarative semantics of the program and lets the user navigate and explore the inference process. It enables the user to use all her knowledge to quickly find the problem and to learn about the program at the same time. An Explorative Debugger is not only a tool to identify the cause of an identified problem but also a tool to try out a program and learn about its 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.

An explorative debugger is a rule browser that:

- Shows the inferences a rule enables.
- Visualizes the logical/semantic connections between rules and how rules work together to arrive at a result. It supports the navigation along these connections[8].
- It allows to further explore the inference a rule enables by digging down into the rule parts.
- Is integrated into the development environment and can be started quickly to try out a rule as it is formulated.

The interested reader finds a complete description of the explorative debugging paradigm and one of its implementation in (Zacharias and Abecker, 2007).

6.3 The Role of Automatic Debugging

Section 6.1 argued that automatic debugging cannot form the exclusive debugging support in the context of the architecture described here. It can, however, play a supportive role for manual debugging systems. The techniques of automatic debugging (and in particular those of why not explanations (Martincic, 2001; Chalupsky and Russ, 2002) can be harnessed to tackle the problem of error reporting (see section 3), to create an initial explanation for the failure of a rule base to return any result to a query. In this way it can play a role similar to that of stack traces in imperative programming - to give a starting point for the developer driven debugging of the system.

7 DISCUSSION

This section adresses two objections against ideas in this paper often encoutered by the authors.

7.1 The Declarativety of Rules

Rules (and declarative languages in general) promise that knowledge represented in this way can be more easily and automatically reused in different contexts, even in those not envisioned during the rule base's creation (McCarthy, 1959):

> Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider

[8]Logical connections between rules are static dependencies formed for example by the possibility to unify a body atom of one rule with the head atom of another. Other logical connections are formed by the prooftree that represents the logical structure of a proof that lead to a particular result

context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because the logical consequences of collections of facts can be available.

Some readers may misunderstand this paper's insistence on visibility - the showing of the interactions between rules - to mean that the authors deny this property and want rule bases to have a rigid/developer defined structure. This however, is not the authors intention. The argument for the importance of visibility is as follows:

1. Only a rule base tested extensively has a chance to work correctly in novel situations.
2. Many of the bugs uncovered during testing will be caused by errors affecting the interaction between rules. Examples for such errors are rules using different attributes to represent the same thing or rules reflecting different interpretations of a class.
3. Supporting the diagnosis of such errors or preventing their introduction into a rule base therefore requires the visbility of rule interactions.

7.2 Comparing Modeling and Programming

This paper frequently draws on comparisons between modeling knowledge as rules and the programming of imperative programs as arguments. Some readers may object to this on the grounds that programming and knowledge modeling are fundamentally different activities and that hence comparisons to programming cannot inform better support for knowledge modeling activities. However, in the end the creation of a knowledge based system is the engineering of a computer system. And in the final creation of this computer system the development of rule based systems encounters many of the same problems as the development with other programming paradigms - such as the problem of identifying the error causing a bug. The building of a knowledge based system is the creation of a computer program in a particular way, but it's the creation of a computer program nonetheless.

8 CONCLUSIONS

Current tools support for the development of rule based knowledge based systems fails to address many common problems encountered during this process. An architecture that combines integrated development, debugging and testing supported by test coverage metrics, visualization and anomaly detection heuristics can help tackle this challenge. The principles of interactivity, declarativity, visibility and modularization can guide the instantiation of this architecture in concrete tools. The novel paradigm of explorative debugging together with techniques from the automatic debugging community can form the robust basis for debugging in this context.

Such a complete support for the development of rule bases is an important prerequisite for Semantic Web rules to become a reality and for business rule systems to reach their full potential.

ACKNOWLEDGEMENTS

Research reported in this paper was partly funded by Vulcan Inc. under Project Halo, and supported by Ontoprise GmbH by giving free access to software for project Online Forecast.

REFERENCES

Baroff, J., Simon, R., Gilman, F., and Shneiderman, B. (1987). Direct manipulation user interfaces for expert systems. pages 99–125.

Brna, P., Brayshaw, M., Esom-Cook, M., Fung, P., Bundy, A., and Dodd, T. (1991). An overview of prolog debugging tools. *Instructional Science*, 20(2):193–214.

Chalupsky, H. and Russ, T. (2002). Whynot: Debugging failed queries in large knowledge bases. In *Proceedings of the Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI-02)*, pages 870–877.

Craw, S. and Boswell, R. (2000). Debugging knowledge-based applications with a generic toolkit. In *ICTAI*, pages 182–185.

Craw, S. and Sleeman, D. (1996). Knowledge based refinement of knowledge based systems. Technical report, The Robert Gordon University.

Decker, S., Erdmann, M., Fensel, D., and Studer, R. (1999). Ontobroker: Ontology-based access to distributed and semi-structured unformation. In *Database Semantics: Semantic Issues in Multimedia Systems*, pages 351–369.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

Dershowitz, N. and Lee, Y.-J. (1987). Deductive debugging. In *SLP*, pages 298–306.

Ginsberg, A. (1988). *Automatic Refinement of Expert System Knowledge Bases*. Morgan Kaufmann Publishers.

Grossner, C., Gokulchander, P., Preece, A., and Radhakrishnan, T. (1994). Revealing the structure of rule based systems. *International Journal of Expert Systems*.

Gupta, U. (1993). Validation and verification of knowledge-based systems: a survey. *Journal of Applied Intelligence*, pages 343–363.

Jacob, R. and Froscher, J. (1990). A software engineering methodology for rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):173–189.

Kifer, M., de Bruijn, J., Boley, H., and Fensel, D. (2005). A realistic architecture for the semantic web. In *RuleML*, pages 17–29.

Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843.

Martincic, C. J. (2001). *Mechanisms for answering "why not" questions in rule- and object-based systems*. PhD thesis, University of Pittsburgh. Adviser-Douglas P. Metzler.

McCarthy, J. (1959). Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London. Her Majesty's Stationary Office.

Mehrotra, M. (1991). "rule groupings: a software engineering approach towards verification of expert systems". Technical report, NASA Contract NAS1-18585, Final Rep.

Morik, J.-U. K. K. and Emde, W. (1993). *Knowledge Acquisition and Machine Learning*. Academic Press, London.

Naish, L. (1992). Declarative diagnosis of missing answers. *New Generation Comput.*, 10(3):255–286.

Ourston, D. and Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 815–520.

Pereira, L. M. (1986). Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210.

Preece, A. D. and Shinghal, R. (1994). Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–701.

Preece, A. D., Talbot, S., and Vignollet, L. (1997). Evaluation of verification tools for knowledge-based systems. *Int. J. Hum.-Comput. Stud.*, 47(5):629–658.

Raedt, L. D. (1992). *Interactive Theory Revision*. Academic Press, London.

Richards, R. M. B. (1991). First-order theory revision. In *Machine Learning: Proceedings of the Eighth International Workshop on Machine Learning*, pages 447–451.

Ruthruff, J. and Burnett, M. (2005). Six challenges in supporting end-user debugging. In *1st Workshop on End-User Software Engineering (WEUSE 2005) at ICSE 05*.

Ruthruff, J., Phalgune, A., Beckwith, L., and Burnett, M. (2004). Rewarding good behavior: End-user debugging and rewards. In *VL/HCC'04: IEEE Symposium on Visual Languages and Human-Centric Computing*.

Saitta, L., Botta, M., and Neri, F. (1993). Multistrategy learning and theory revision. *Machine Learning*, 11(2):153–172.

Shapiro, E. Y. (1982). *Algorithmic program debugging*. PhD thesis, Yale University.

Stumptner, M. and Wotawa, F. (1998). A survey of intelligent debugging. *AI Commun.*, 11(1):35–51.

Tsai, W.-T., Vishnuvajjala, R., and Zhang, D. (1999). Verification and validation of knowledge-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):202–212.

Waterman, D. A. (1968). *Machine Learning of Heuristics*. PhD thesis, Stanford University.

Wilkins, D. (1990). Knowledge base refinement as improving and incorrect, inconsistent and incomplete domain theory. *Machine Learning*, 3:493–513.

Wogulis, J. and Pazzani, M. (1993). A methodology for evaluating theory revision systems: results with audrey ii. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 332–337.

Zacharias, V. (2007). Visualization of rule bases - the overall structure. In *7th International Conference on Knowledge Management - Special Track on Knowledge Visualization and Knowledge Discovery*.

Zacharias, V. and Abecker, A. (2007). Explorative debugging for rapid rule base development. In *Proceedings of the 3rd Workshop on Scripting for the Semantic Web at the ESWC 2007*.