Zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften (Dr. rer. pol.)
von der Fakultät für Wirtschaftswissenschaften
der Universität Fridericiana zu Karlsruhe
genehmigte Dissertation

Tool Support for Finding and Preventing Faults in Rule Bases

von

M.S./USA Valentin Zacharias

Tag der mündlichen Prüfung: 18.12.2008
Referent: Prof. Dr. Rudi Studer
Korreferent: Prof. Dr. Karl-Heinz Waldmann

Abstract

The starting point for this thesis was the apparent contradiction between the perception of rule bases as simple to create and the experience of rule bases as hard to debug and difficult to create without faults. This contradiction was analyzed using data from a survey of developers, experiments and experiences from three rule base development projects. From the aggregation of this analysis using the developed fault lifecycle model, testing, debugging, static analysis and visualization were chosen as concrete and particularly promising approaches for tackling this fault prevention challenge. Each of these areas was further examined within the context of this thesis.

In the area of **testing**, a formal account of the notions of the test entities was developed, a novel test coverage measure based on least general generalization was conceived and a testing framework based on these notions was implemented and evaluated. The evaluation over more than 100 hours showed the usefulness and importance of the concepts and their implementation.

To better support the **debugging** of rule bases, Explorative Debugging was proposed as a novel and purely declarative debugging paradigm for these systems. An experiment comparing Explorative Debugging to the state of the art procedural debugging paradigm showed a significant improvement in the time needed to identify faults, while the accuracy increased.

To improve the **static analysis** for fault detection, an anomaly detection framework for F-logic was developed. This framework is the first of its kind for F-logic and it includes the first implementation of static type checking for F-logic. It is mostly implemented in F-logic itself, integrated into a rule engineering environment and easily extensible.

Finally, to support users in understanding the rule base and the consequences of changes to it, a novel approach to the **visualization** of the structure of the rule base was developed. The novelty of this approach lies in the use of runtime rule interactions to show the overall structure of the rule base.

Zusammenfassung

Ausgangspunkt dieser Arbeit war der scheinbare Widerspruch zwischen der allgemeinen Meinung, dass Regelbasen einfach zu erstellen sind, und der Beobachtung, dass diese tatsächlich nur schwer fehlerlos zu erstellen und zu debuggen sind. Mittels unterschiedlicher Experimente, einer Umfrage unter Entwicklern und Erfahrungen aus drei Projekten zur Erstellung von Regelbasen wurde dieser Widerspruch analysiert. Unter Zuhilfenahme des hierfür entwickelten Fehler-Lebenszyklus-Modells wurden die Ergebnisse der Analyse aggregiert. Dadurch wurden Ansätze zu Tests, Debugging, Statischer Analyse und Visualisierung als besonders vielversprechend identifiziert. Jeder dieser Ansätze wurde im Rahmen dieser Arbeit genauer untersucht.

Um das Testen von Regelbasen besser zu unterstützen, wurden formale Konzeptbeschreibungen von Testentitäten entwickelt, ein Testvollständigkeitsmaß erstellt sowie auf deren Basis ein Testrahmenwerk implementiert und evaluiert. Dessen mehr als 100 stündige Evaluation zeigte die Nützlichkeit und Wichtigkeit dieser Konzepte und ihrer Implementierung.

Zur besseren Unterstützung des Debuggings von Regelbasen, wurde Exploratives Debugging als neues und rein deklaratives Debugging-Paradigma vorgeschlagen. In einem mit dem Stand der Technik vergleichenden Experiment konnte eine signifikante Zeitersparnis bei höherer Genauigkeit zur Identifikation von Fehlern nachgewiesen werden.

Zur Verbesserung der statischen Analyse wurde ein Rahmenwerk zur Erkennung von Anomalien für F-Logik entwickelt. Dieses Rahmenwerk ist für F-Logik das erste seiner Art und enthält die erste Implementierung einer statischen Typprüfung für F-Logik.

Zuletzt wurde ein neuer Visualisierungsansatz für Regelbasen entwickelt, der dem Nutzer hilft, die Regelbasis leichter zu verstehen und Folgen von Änderungen nachvollziehen zu können. Die Neuheit dieses Ansatzes liegt in der Verwendung von Laufzeit-Regelinteraktionen, um die Gesamtstruktur der Regelbasis anzuzeigen.

Acknowledgements

My special thanks go to my doctoral thesis supervisor Prof. Dr. Rudi Studer who gave me invaluable advice and the freedom to pursue my studies. I am also indebted to Dr. Andreas Abecker, particularly for his thousands of helpful comments to early versions of this text and his help in finding the time to write it.

I also want to thank Prof. Dr. Karl-Heinz Waldmann for taking on the responsibility as second examiner and Prof. Dr. Werner Rothengatter and Prof. Dr. Stefan Tai for their role in the exam.

I want to thank many colleagues and students for their feedback and their help and collaboration in realizing this work. I'm particularly grateful to Imen Borgi, Michael Erdmann, Mark Greaves, Anthony Jameson, Frank Kleiner, Kai Kühn, Torsten Pfirrmann, Thomas Schweitzer and Mrudula Sreekanth. This work also wouldn't have been possible without the support from FZI, Ontoprise, Vulcan Inc. and Daimler Chrysler.

I wrote this dissertation while working at the FZI research center and I want to thank all my colleagues for making all these years such an enjoyable experience. In particular I want to thank Mark Hefke, Alexander Mädche, Boris Motik, Hans-Jörg Happel, Peter Wolf, Carsten Holtmann, and Ljiljana Stojanovic.

I particularly want to thank my parents who made all this possible and supported me all the time. Also my heartfelt thanks to Silke and Simone, without them I wouldn't have had the courage to either start or finish this work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Rule based systems are computer programs built according to a paradigm that stipulates to represent the program in IF-THEN structures (rules), to strive to represent the program declaratively and to use an inference engine that automatically combines the rules based on the task at hand. With rule based systems comes the hope to build programs in a simpler, more flexible and more modular way. These systems are assumed to be simpler to create, because as (mostly) declarative languages they free the developer from worrying about the *how* of the computation, because the IF-THEN structure naturally resembles the way humans communicate a large part of their knowledge and because the basic structure of rules is very simple and easy to understand. The promise of increased flexibility and modularity rests on the observation [107] that declarative sentences are true in a wider context than program statements can be used.

Rule based systems have their roots in the research on Artificial Intelligence and Expert Systems. In their development over more than 30 years hundreds of different rule languages and environments have been developed and have been used to create an untold number of computer programs, performing a variety of tasks from computer games, over rating of credit risks, to the diagnosis of bacterial infections. Recently, with the large-scale practical use of business rule systems [150] and the interest of the Semantic Web community in rule languages [91], the popularity of these kinds of system has again started to increase.

Preventing, finding and removing faults in rule bases is - like in all software development - and important and challenging task. Indeed, a survey of developers found related problems such as debugging, determining test coverage and determining completeness of the created rule base among the issues most hindering the development of rule based systems (see section 3.1.1). Moreover, in three observed development projects, finding faults posed such difficulties that it almost negated the advantages of simplicity associated with rule based systems. A survey of developers also found, that while developers see rule bases as indeed simpler to create, understand and maintain, they say that debugging rule bases is actually more difficult than the debugging of comparable procedural or object oriented programs (see figure 1.1).

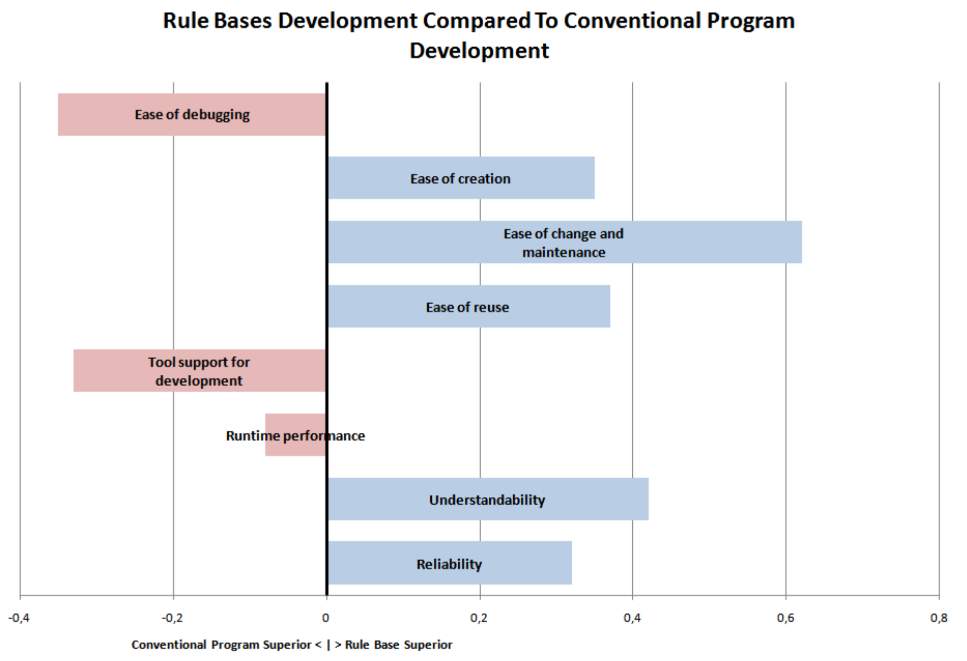


Figure 1.1: *The aggregated results from the survey questions asking participants to compare rule base and 'conventional' software development*

Changes in the way how rule bases are built are also posing challenges and are offering new opportunities for the development process. The continued rise of end-user programming, the embedded nature of many modern rule bases, the rise of agile methods and the increasing interest in the Semantic Web [16] are forces influencing modern rule base development and how faults can be prevented, identified and removed.

From these observations follow the overall questions of this thesis: why are faults in rule bases hard to find? How is that influenced by the intrinsic properties of rule languages and by modern rule base development processes? And finally and most importantly - what can be done about it?

1.2 Approach

The overall goal of this thesis is an improvement in the effectiveness of fault prevention, identification and removal for rule based systems. Towards this end it was carefully analyzed why this is currently problematic; how the difficulty in finding faults can be explained. This analysis results in a number of hypotheses explaining where practical problems lie; these hypothesis are then used to derive areas that can most profit from better tool support. For some of these areas several novel tools were then conceived, developed and experimentally evaluated.

The scientific approach of this thesis rests on seven pillars. These pillars comprise of qualitative observation in two rule base development projects, systematic observation and experiments in another project and one dedicated experiment. An important role is also played by literature research and a survey of developers of rule based systems. An overview of the approach is shown in figure 1.2; the remainder of this section is dedicated to a short explanation of all elements in this figure.

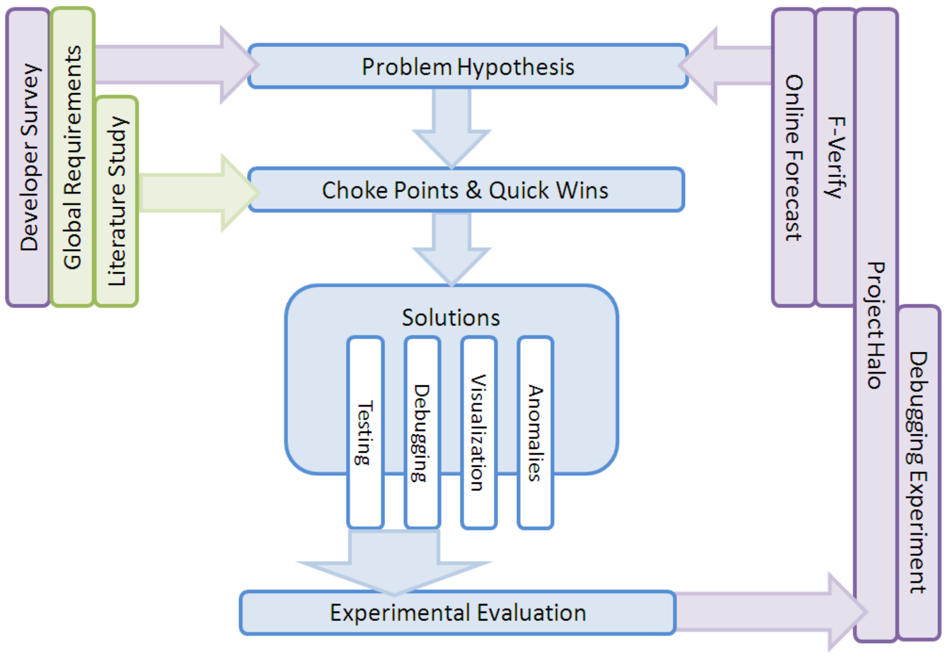


Figure 1.2: *Methodological approach of this thesis*

The fault identification challenge was identified based on qualitative observation in the two projects F-Verify and Online Forecast. Some hypotheses for its explanation were also derived in these projects. **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 a rule based system. The goal of the project **F-verify** was to create a rule base as support for verification activities. 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. These two projects are described in detail in the section 3.1.3.

A third project, Project Halo, was used to derive further requirements and as a test bed for the experimental evaluation for the developed testing tools. **Project Halo** 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. Project Halo is described in more detail in section 3.1.2.

A **survey of developers** of rule based systems (described in section 3.1.1) is the fourth input to the identification of the fault identification challenge and its explanation.

Further input comes from an exploration of **global requirements** that follow from the shifting nature of rule base development (see section 3.2.4) and an extensive **literature study** of relevant - often older and inactive - research (see, in particular, section 5.4).

Based on the aggregation of the results from the analysis four choke points were identified for further study: debugging, testing, static analysis and mistakes stemming from limited knowledge about the overall structure of the rule base. One tool each was developed to address these areas.

A regression testing framework for F-logic was conceived and developed to better support the testing challenge. It contains graphical interfaces to enable end user programmers to create tests and further a novel test coverage metric for rule base was developed to support users in creating complete test sets. This testing framework was evaluated within Project Halo.

Explorative Debugging was developed as a new debugging paradigm to improve the efficiency of fault identification in rule based systems. Novel ways to understand and show the inference process in rule bases were developed in support of this debugger. Explorative Debugging and its components were evaluated in a dedicated **debugging experiment**. In this experiment random faults were introduced into the rule base and two different kinds of debugger were compared in their efficiency in finding these faults.

To support users in understanding the overall structure of the rule base a new visualization approach was developed. This approach uses the runtime rule interactions of the rule during query answering and tests to derive the relations between the rules. Finally a collection of anomaly heuristics was created to support the static analysis of rule bases, also profiting from the availability of ontologies.

1.3 Contribution

The main contribution of this thesis is the *comprehensive* treatment of the question what can be done about the difficulty of fault prevention, identification and removal in rule based systems. This answer is based on experiences and empirical data from multiple project, experiments and a survey of developers. The main result is an - albeit incomplete - overall answer to the question "What can be done about the debugging challenge of rule based systems?"

Within this comprehensive framework a number of individual, delimitable contributions were made. Conceptually Explorative Debugging (section 5.1) was developed as a new way to approach the debugging of rule based systems. It includes the novel notions of mutation extended depends-on graph and mutation extended prooftree, new concepts that can be used for the declarative debugging of rules. Its realization as an open source debugger includes the implementation of these concepts and is unique in many aspects. A second implementation of the Explorative Debugging paradigm is more limited in its functionality, but is included in a commercial development environment and was the first graphical debugger for F-logic.

In the context of testing, the first testing framework for F-logic was designed and implemented; a new test coverage measure for normal logic programs was conceived. The evaluation of this framework showed its suitability for end users.

In further conceptual work a new approach for the overall visualization of rule bases was conceived and implemented. It is unique because of its utilization of actual runtime rule interactions for the creation of the visualization.

In a further development the first set of anomaly detection heuristics was implemented in and for F-logic. The verification support for the DarkMatterStudio system (section 3.1.2) was conceived and implemented as part of this work.

New empirical data was created in particular through a survey of developers of rule based systems, in a debugging experiment and through action research in two rule base development projects.

A further contribution is the only current state of the art survey in rule debugging - bringing together work from very different and sometimes stale areas of research (section 5.4), and the only systematic account of the syntax and semantics of F-logic/LP-O; the commercially most important dialect of F-logic (section 2.2).

1.4 Overview

This thesis is structured in six main parts: (2) Fundamentals, (3) Analysis and Design, (4) Testing, (5) Debugging, (6) Visualization and (7) Anomalies.

1. The second chapter, Fundamentals, gives a precise characterization of the concepts necessary for understanding this text. It introduces normal logic programs as the theoretical basis for all subsequent chapters. F-logic and horn rules on top of RDF - two formalisms which are used in implementations presented in this thesis - are described and it is shown how they can be reduced to normal logic programs.

2. In the beginning of the Analysis and Design chapter the main body of empirical evidence is introduced. This chapter starts with a description of the three projects that formed the basis for the identification and explanation of the verification challenge. Also the survey of developers and its results are presented. Based on the results from the survey, the experiences and the data from the projects a number of explanation hypotheses are developed, aggregated and then used to decide on the approach taken.

3. The testing chapter introduces a regression testing framework for F-logic and describes its subject matter tailored user interface. The chapter further describes an novel test coverage measure for normal logic programs and the evaluation of the test framework.

4. The debugging chapter first gives a extensive overview over the state of the art in debugging rule based systems. Based on the deficiencies in the existing systems and the requirements and design principles identified in chapter 3 Explorative Debugging is presented. The description of Explorative Debugging first focuses on the conceptual level and then describes two implementations and an evaluation.

5. The visualization chapter discusses how the runtime interactions of the rules can be used to derive an overall picture of the rule base that can aid the user in making correct changes.

6. The seventh chapter introduces a set of anomaly heuristics and their embedding into an application that can support the user in finding

faults through static analysis of the source code.

In the last chapter the conclusions summarize the content of the thesis, its contribution and discuss possible future work.

1.5 Publications

The work presented in this thesis is partly published in the following publications:

- Valentin Zacharias, Imen Borgi: **Exploiting Usage Data for the Visualization of Rule Bases**, Proceedings of the 3rd International Semantic Web User Interaction Workshop, SWUI 2006 at the International Semantic Web Conference (ISWC), 2006.
- Mark Hefke, Valentin Zacharias, Ernst Biesalski, Andreas Abecker, Qingli Wang, Marco Breiter: **An extendable Java Framework for Instance Similarities in Ontologies**, Proceedings of the 8th International Conference on Enterprise Information Systems, 2006.
- Valentin Zacharias, Andreas Abecker: **Explorative Debugging For Rapid Rule Base Development**, Proceedings of the 3rd Workshop on Scripting for the Semantic Web at the 4th European Semantic Web Conference (ESWC), 2007.
- Valentin Zacharias, Andreas Abecker: **On Modern Debugging For Rule-Based Systems**, Proceedings of the The Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE), 2007.
- Valentin Zacharias: **Visualization of Rule Bases - The Overall Structure**, Proceedings of the 7th International Conference on Knowledge Management (I-Know), Special Track on Knowledge Visualization and Knowledge Discovery, 2007.
- Valentin Zacharias: **The Agile Development of Rule Bases**, Proceedings of the 16th International Conference on Information Systems Development (ISD), 2007.
- Valentin Zacharias: **Rules As Simple Way to Model Knowledge - Closing the Gap between Promise and Reality**, Proceedings of the 10th International Conference on Enterprise Information Systems (ICEIS), 2008.
- Valentin Zacharias: **Development and Verification of Rule Based Systems - a Survey of Developers**, Proceedings of the International RuleML Symposium on Rule Interchange and Applications, 2008.

- Andreas Abecker, Valentin Zacharias: **Comprehensive Developer Support for Rule-Based Programming**, Proceedings of the eChallenges Conference, 2008.
- Valentin Zacharias: **The Debugging of Rule Bases**, to appear in Handbook of Research on Emerging Rule-Based Languages and Technologies, IGI Global, Hershey (USA) 2009.
- Valentin Zacharias: **Tackling the Debugging Challenge of Rule Based Systems**, to appear in Enterprise Information Systems X, Springer-Verlag, Berlin (Germany) 2009.

Chapter 2

Fundamentals

This chapter introduces and defines the formal basis for this thesis; it introduces the rule languages used for definitions and in implementations. Altogether three rule languages are introduced.

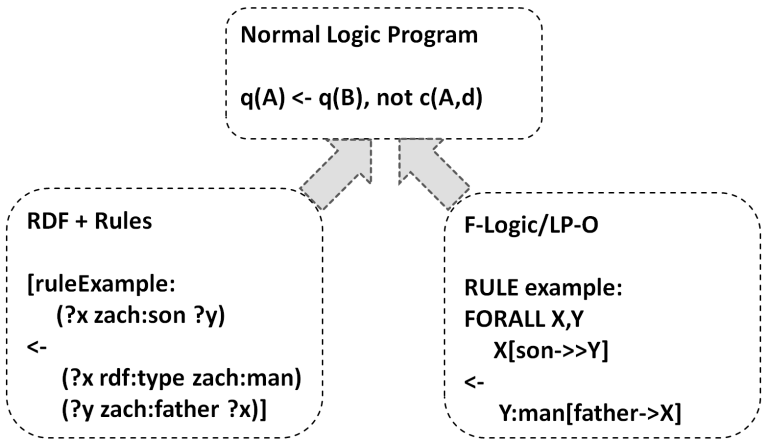


Figure 2.1: *The three rule languages discussed in this chapter*

The three languages and their role are (see also figure 2.1):

1. **Normal Logic Programs** [66, 164] are used as the common basis for all chapters. Definitions are given with respect to this language to make them applicable as widely as possible. Normal logic programs and their semantics are described in section 2.1

2. **F-logic/LP-O** [92, 3] is used in some implementations, in particular in the testing and anomaly chapters. Doing these implementations in F-logic/LP-O was necessary to integrate the applications into a commercial ontology engineering environment. F-logic/LP-O is a syn-

tactic variant of normal logic programs in the sense that every F-logic program can be transformed into a normal logic program. The F-logic Syntax as well as the details of this transformation are described in section 2.2.

3. **RDF + Rules** [29, 30], a rule language on top of RDF is used in other implementations and is described in section 2.3. This rule language was chosen because of its importance to the Semantic Web. This rule language too can be transformed into a normal logic program.

In summary: because of project and implementation context all implementations in this thesis were done for either F-logic/LP-O or RDF Rules; both variants of normal logic programs. To describe all concepts in a unified way and to make them applicable in a wider context, normal logic programs are used as basis for the conceptual work in this thesis.

In this section the three languages and the transformations to normal logic programs are described. Considerable time is spend on the description of the F-logic/LP-O and its transformation to normal logic programs - to the authors knowledge the only such description of the syntax and semantics of an actual F-logic implementation.

Not everything in this chapter is needed to understand the remaining chapters; the precise characterization of F-logic and the description of the transformation to normal logic programs is a contribution in its own right and are needed only to establish normal logic programs as common basis. A reader will be able to understand the remaining chapters with an understanding of the least Herbrand model for normal logic programs and the ability to read simple F-logic and RDF rules, something that can be learned from the examples in section 2.2.1 and at the beginning of section 2.3.4.

2.1 Normal Logic Programs

A *normal logic program* is a finite set of rules. An example for a rule would be

$$p(A) \leftarrow q(B), \neg\, r(A, d)$$

The part to the left of the $\leftarrow$ symbol is the *goal*, *head* or *consequent* of the rule; the part to the right the *body*, *antecedent*, consisting of the *subgoals*. A rule without a body is called a *fact*; a rule without head a *query*. A rule is read as *if something then something else* and encodes conditional knowledge. All the rules in a rule set are combined to derive answers to queries.

The following is a simple example rule base:

$$\begin{aligned}
&father(A) \leftarrow male(A), child(A,X)\\
&ancestor(A,X) \leftarrow child(A,X)\\
&ancestor(A,X) \leftarrow ancestor(A,Y), ancestor(Y,X)\\
&male(peter)\\
&child(peter, mike)
\end{aligned}$$

The first rule can be read as *A is a father if A is male and A has a child*. All words starting with an uppercase letter, like the A in this rule, stand for variables. The second and third rule define the ancestor relationship - stating that A is the ancestor of X, if X is her child; and that A is the ancestor of X if A is the ancestor of Y and Y is the ancestor of X. Two facts state that Peter is male and has the child mike. A query to this rule base could ask for all instances known to be father:

$$\leftarrow father(A)$$

With the facts and rules above the evaluation of this query would return $A = Peter$.

This section gives a precise characterization of the syntax and the semantics of normal logic programs. It starts with the syntax (Section 2.1.1) and the definition of unification (Section 2.1.2). Afterwards the semantics of logic programs is described without (Section 2.1.3) and with considering negation (Section 2.1.4). Stratifiability is presented as a property of normal logic programs that determines whether the presented semantics can be applied (Section 2.1.5). In the last part built-ins and their integration into normal logic programs are presented (Section 2.1.6).

2.1.1 Syntax

The alphabet of a normal logic program consists of:

- A set of variables, Var, of all finite alphanumeric strings.
- A set of predicate symbols, $Pred$, of all finite alphanumeric strings. There is a mapping $arity : Pred \rightarrow \mathbb{N}^+$ from the set of predicates to the set of all positive integers greater than 0. If there is a mapping for a predicate p, $arity(p) = n$, then p is an n-ary predicate.
- A set of function symbols, $Func$, of all finite alphanumeric strings. There is a mapping $arity : Func \rightarrow \mathbb{N}_0$ from the set of function symbols to the set of all positive integers. If there is a mapping for a function f, $arity(f) = n$, then f is an n-ary function symbol. We also call all 0-ary function symbols constants.

- Auxiliary symbols and logical connectives such as $\leftarrow$,$\neg$,,,(,)[1]

The sets Var, $Pred$ and $Func$ are disjoint. Throughout this text the following conventions are observed: variables are written as strings beginning with an uppercase character. Both functions and predicates are written beginning with a lowercase character.

A **term** is defined inductively as:

- A variable is a term.
- A 0-ary function symbol (a constant) is a term.
- If $(a_1, ..., a_n)$ are terms and f is an n-ary function symbol, then $f(a_1, ..., a_n)$ is a term.

A **ground term** is a term that contains only constants and function symbols. An **atom** $p(a_1, ..., a_n)$ is defined as a n-ary predicate symbol p and a list of n terms as arguments. A **ground atom** is an atom with only ground terms as arguments. A **literal** is an atom that may be negated, it is called **positive literal** when the atom is not negated, **negative literal** if it is. A negative literals is written as **not**$p(a_1, ..., a_n)$ A **ground literal** consists of a ground atom.

A **rule** is a structure of the form $A \leftarrow L_1, ..., L_n$ where A is an atom and $L_1, ..., L_n$ is a set of literals. The atom is called the **head** and the set of literals is called the **body** of the rule. A rule expresses conditional knowledge and is read as A if L_1 and L_2 and .. and L_n. A rule without a body is expressing unconditional knowledge and is called a **fact**. A **goal** or **query** is a rule consisting only of a body.

A **normal logic program** is a set of rules.

2.1.2 Substitution and Unification

A **substitution** θ is a set $\theta = \{V_1/t_1, ...V_n/t_n\}$, where $V_1, ..., V_n$ are distinct variables and $t_1, ..., t_n$ are terms such that $t_x \neq V_x$ for all x in $1...n$. An element V_x/t_x of a substitution is called a **binding** for V_x. A substitution where all $t_1, ..., t_n$ are ground terms is a **ground substitution**.

An **instance** of a rule or atom $E\theta$ is defined as the simultaneous replacement of all variables V_x in the rule/atom with the terms in their bindings; i.e. V_x is replaced by t_x where $V_x/t_x \in \theta$. This process is called the **application of** θ **to** E. $E\theta$ is a **ground instance** of E iff it contains no variables.

[1]Note that komma is used both as auxiliary symbol and to delimate the elements

The **composition** of two substitutions $\theta = \{V_1/t_1, ...V_n/t_n\}$ and $\sigma = \{W_1/u_1, \quad ..., W_m/u_m\}$ is defined as $\theta\sigma = \{V_1/t_1\sigma, ..., V_n/t_n\sigma, W_1/u_1, ..., W_m/u_m\}$, where identity substitutions (i.e. $V_k/t_k\sigma$ where $V_k = t_k\sigma$) and all W_k/u_k, where W_k is equal to some V_j are removed. The application of $\theta\sigma$ to a rule E is exactly the same as the application of first θ and then σ; i.e. $E\theta\sigma = (R\theta)\sigma$.

Two rules or atoms E and F are **variants** of each other, iff substitutions θ and σ exist, such that $E\theta = F$ and $F\sigma = E$.

Two rules or atoms E and F are **unifiable**, iff there is a substitutions θ such that $E\theta = F\theta$, θ is the **unifier** of E and F. A substitution θ is **more general** than another substitution σ iff there is a substitution τ such that $\theta\tau = \sigma$. A unifier of two expressions is called **most general unifier mgu** iff it is more general than all other unifiers of these two rules or atoms. However, note that the mgu is not unique and that there exist unifiers θ, σ such that $\theta \neq \sigma$, θ is more general than σ and σ is more general than θ.

2.1.3 Semantics

This section describes the semantics first for rules restricted to only positive literals in the rule body. The semantics are extended with negations and built-ins in the following sections.

The **Herbrand base** HB is the set of all atoms that can be formed with the set of constants, functions symbols and predicates. A **Herbrand interpretation** I is a subset of the Herbrand base. We define the truth of a rule under an Herbrand interpretation I as follows: A rule $R = A \leftarrow L_1, ..., L_n$ is $true$ under I iff for all substitutions θ such that $L_1\theta \in I, ..., L_n\theta \in I$ it holds that $A\theta \in I$. This implies that in particular a fact F is $true$ under I iff all its ground instances $F\theta \in HB$ are in I.

If a rule E is true under an Herbrand interpretation I, we say that I **satisfies** E. A Herbrand Interpretation I is a **Herbrand model** for a logic program P iff I satisfies all rules $E \in P$. A set of rules S is a **consequence** of a logic program P iff all models for P are also models for S; this entailment relation is denoted as $P \models S$.

The **least Herbrand model** LHM of a logic program P is the intersection of all models of P. The least Herbrand model $LHM(P)$ is also a model for P [31]. The least Herbrand model is one distinguished model that defines the consequences of a logical program. The **result of a query** $G :\leftarrow L_1, ..., L_n$ to a logic program P is defined as the set of all ground instances of G that are satisfied by the least Herbrand model of P.

2.1.4 Semantics with Negation

The previous section restricted the discussion of the semantics to rules with only positive literals. In this section the **Closed World Assumption CWA** is adopted [137] to allow conclusions from negative facts. The CWA states that when a ground fact is not known to be true, it is assumed to be false.

Negative facts consisting of an atom A are written as $\neg A$. The notation $|\neg A|$ is used to describe the positive equivalent to negative facts, e.g. $|\neg pred(a, b)| = pred(a, b)$.

The definition for the truth of a rule R under a Herbrand interpretation I is extended by stating that a negative literal L_x is $true$ if its positive counterpart $|L_x|$ isn't. I.e. a rule $R = A \leftarrow L_1, ..., L_n$ is $true$ under I iff for all substitutions θ and

$$\text{all literals } L_i \in L_1, ...L_n \begin{cases} L_i\theta \in I, \text{ if } L_i \text{ is a positive literal} \\ |L_i|\,\theta \notin I, \text{ if } L_i \text{ is a negative literal} \end{cases}$$

A negative fact F is $true$, iff for all substitutions θ it holds that $|F|\,\theta \notin I$.

Based on this definition of truth, consequence under CWA can be defined: A set of rules and facts S is a **consequence** of a logic program P, iff all models for P are also models for S, this is written as $P \models_{CWA} S$.

Defining a distinguished model is more difficult, because logic programs with negation do not have the property that the intersection of all models is itself always a model; for many programs there is not even a unique least model. Different semantics have been proposed and are employed to deal with this problem (e.g. [164, 66]). Here the semantics based on globally stratified models is presented, a straight forward extension of the least model semantics that was used in many implementations described later.

The intuition behind stratified semantics is, that it is possible to find an ordering of the predicates in a program that can be used to identify a distinguished model that usually matches the intuition of the person that created a program. An added advantage is that this ordering of the predicates can be used in the efficient computation of answers to queries against such a program. The big disadvantage is that not all programs are globally stratifiable; programs have to be changed or a different semantics has to be used for programs not within this class.

We define the **ground expansion** P_0 of a logic program P as the set of ground instantiations that can be formed from the rules in the logic program P and the Herbrand base HB. We define the relation $\leftharpoonup$ between two elements A and B of the Herbrand base as: $A \leftharpoonup B$ iff there is a ground instance in P_0 where A appears in the head and B (positively or negatively) in the body. $A \leftharpoondown B$ can be read as: B is used to conclude A. We say that a $A \hookleftarrow B$ iff there is a sequence such that $A = E_1 \leftharpoondown E_2, E2 \leftharpoondown E3, ..., E_{n-1} \leftharpoondown E_n = B$ and at least one of the E_i appears negatively in the rule body. $A \hookleftarrow B$ is read as: B has a higher priority than A and can be understood as: B should be computed before A. If M_1 and M_2 are two Herbrand models, then M_1 is preferable to M_2, iff for each element $E \in (M_1 - M_2)$ there exists an element $F \in (M_2 - M_1)$ such that $E \hookleftarrow F$. Or, intuitively, a model is preferable to another model, iff it can be obtained by replacing higher priority atoms with lower priority ones. A Herbrand model HM_p for a logic program P is **perfect**, iff there is no model for P that is preferable to HM_p. This perfect model defines the result of queries to the program in the same way the least Herbrand model does for programs without negation. A perfect model exists for all **stratified logic programs** [31].

2.1.5 Stratification

An **extended dependency graph** EDG for a logic program P is a directed graph $G = (V, E)$ and a function $f_p : E \rightarrow \{positive, negative\}$. The vertices of the graph are the predicate symbols $Pred$. There is an edge $e = (p, q)$ with $p \in V$ and $q \in V$, iff there is a rule $R \in P$ such that p is in the rule body and q is in the rule head. $f_p(e) = negative$, iff there is at least one rule with q in the head that contains p in a negative literal in the body; otherwise $f_p(e) = positive$.

A **stratification** of a logic program is a partition of $Pred$ into disjoint subsets $S_1, ..., S_n$ such that for each $e = (p, q) \in E$:

$$\text{if} f_p(e) = positive \text{ and } p \in S_i \text{ and } q \in S_j \text{ then } j \geq i.$$
$$\text{if} f_p(e) = negative \text{ and } p \in S_i \text{ and } q \in S_j \text{ then } j > i.$$

A program is **stratifiable**, iff a stratification exists.

2.1.6 Built-ins

Built-ins are special predicates such as $<$, $\neq$ or $\geq$ that are defined neither by rules nor storing their (usually infinite) extension as ground facts; rather built-ins are implemented as procedures that evaluate the truth of the predicate for ground parameters at runtime. Built-ins must appear only in rule bodies, never in rule heads.

Built-ins can be viewed as a special kind of ground facts that are stored in a different way: not explicitly but implicitly in program code; hence the discussion of semantics in the preceding sections stands without needing to be changed. However, because built-ins correspond to infinite relations and because the procedures are usually not able to generate their extensions, safeness constraints need to be imposed on the rules using built-ins. For example an implementation for the built-in $<$ is usually not able to even start enumerating all numbers larger than 3, when called as $<(3, X)$ - it can only evaluate whether the relations holds for two concrete numbers. For each built-in there is a **safeness constant** sc that reflects the minimal number of parameters that have to be bound to ground terms. Often the safeness constant is equivalent to the arity of the built-in (as in the example of $<$), but sometimes it is smaller (e.g. for the $=$ built-in the safeness constant is one). A rule is **safe with respect to built-ins**, iff at least sc of its parameters also appear in positive literals in the rule body. We require a logic program to consist only of rules that are safe in this respect.

2.2 F-logic

F-logic [92] or Frame Logic was developed as an attempt to create a clearly defined declarative semantic for deductive object oriented databases. Two versions of F-logic were defined, one based on first order logic (F-logic/FOL), the other based on logic programming languages (F-logic/LP). Only F-logic/LP is of continued practical relevance and is discussed in this section. This section is further based on the dialect of F-logic/LP implemented in the inference engine Ontobroker. (F-logic/LP-O)[59, 50] - the only commercial implementation of F-logic. F-Logic/LP-O together with the inference engine Ontobroker was used in many implementations described later. A short discussion how the F-Logic/LP and F-logic/LP-O differ is given at the end of this section. F-logic/LP-O is a dialect of normal logic programs in the sense that it can be syntactically transformed to a (subset) of normal logic programs.

This section starts with a short example for the use of F-logic/LP-O, followed by a detailed description of its syntax. The next sections then present the syntactic transformations of F-logic/LP-O into normal logic programs that also define the semantics. The final section gives a short overview of the differences between F-logic/LP-O and F-Logic/LP as defined in [92]. This section is only meant to define and give an overview of F-logic. Readers interested in learning F-logic for practical use should also consult [2, 63].

2.2.1 Example

```
/* facts */
```

```
man::person.
woman::person.
Abraham:man.
Sarah:woman.
Isaac:man[father->Abraham; mother->Sarah].
/* rules */
FORALL X,Y
    X[son->>Y]
    <-
    Y:man[father->X].
FORALL X,Y
    X[son->>Y]
    <-
    Y:man[mother->X].
/* query */
FORALL X,Y <-X:woman[son->>Y] AND Y[father->Abraham].
```

The first part of this example consists of a set of facts. The first two lines state that man and woman are subclasses of person. A subclass of relation means that the signature of the superclass is inherited and that all instances of the subclass are also instances of the superclass.

The next two lines assert that Abraham and Sarah are instances of man and woman respectively. Through inheritance they are then also instances of person. Intuitively the main purpose of the instance-of relation is that rules can be built that apply to all instances of a particular class. In F-logic classes and instances are both terms and a class can itself be again an instance of another class.

The last line of the facts states that Isaac is also a man and that there is a relation *father* from Isaac to Abraham and a relation *mother* from Isaac to Sarah. In F-logic, based on the language from object-oriented programming, this is understood as stating: the application of the method father to the object Isaac yields the result object Abraham.

The rules in the second part of the example derive new information from the given object base. On the evaluation of these rules new relationships between objects, denoted by the method ″son″ are inferred. In English the first rule reads: for all objects X and Y, X has the son Y, if Y is a man and his father is X.

The third part of the example contains a query to the object base. It asks for all the women who have sons whose father is Abraham; both the women and the sons will be returned as result.

2.2.2 Syntax

The alphabet of F-logic/LP-O consists of the following (except for the auxiliary symbols it is identical to the alphabet for normal logic programs):

- A set of variables, Var, of all finite alphanumeric strings.
- A set of predicate symbols, $Pred$, of all finite alphanumeric strings. There is a mapping $arity : Pred \rightarrow \mathbb{N}^+$ from the set of predicates to the set of all positive integers greater than 0. If there is a mapping for a predicate p, $arity(p) = n$, then p is an n-ary predicate.
- A set of function symbols, $Func$, of all finite alphanumeric strings. There is a mapping $arity : Func \rightarrow \mathbb{N}_0$ from the set of function symbols to the set of all positive integers. If there is a mapping for a function f, $arity(f) = n$, then f is an n-ary function symbol. We also call all 0-ary function symbols constants.
- Auxiliary symbols and logical connectives:
  `AND,OR,FORALL,EXISTS,`
  `RULE,<-,->,->>,=>,=>>,.,,,[,],{,},:,@` [2]

Analog to normal logic programs, a term is defined as:

- A variable is a term.
- A 0-ary function symbol (a constant) is a term.
- If $(a_1, ..., a_n)$ are terms and f is an n-ary function symbol, then $f(a_1, ..., a_n)$ is a term.

Note that both classes and instances are terms and that - depending on the statement - a term may be treated both as an instance and a class.

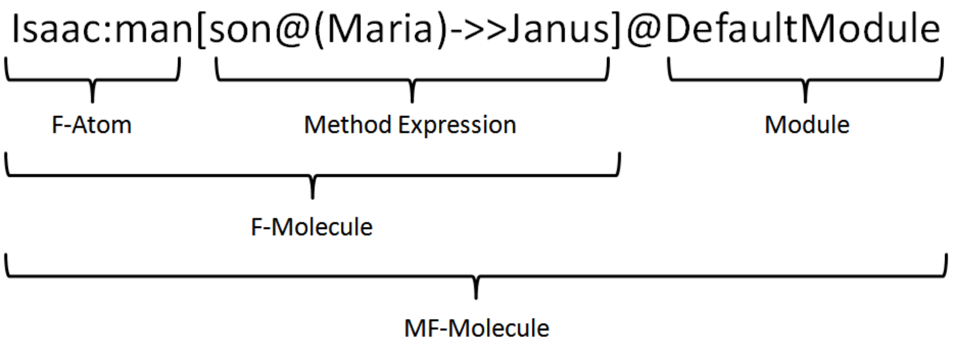


Figure 2.2: *F-Logic syntax terminology*

[2]Note that comma is used both as auxiliary symbol and to delimit the elements

In order to aid the understanding of the F-logic syntax, the most important terms will are introduced in an example in figure 2.2. This example shows a MF-molecule (a 'modularized f-molecule') that states that: Isaac is a man and that the method son with the parameter Maria returns Janus. This information is stated in the module with the name DefaultModule. Note in particular that the symbol '@' serves two purposes - as a delimeter before the module and as a delimiter between the method name and the parameters. An MF-Molecule without the module is called F-Molecule, it consists of an F-atom and zero or more method expressions. It is important to note that the method expressions describe the application of methods to the term in the F-atom preceding them, e.g. in the example the value of the method application son@maria to the instance Isaac is defined.

A **method expression** is a statement of the form `B@{E1...En}~>C`, where B is an n-ary function symbol, C and all E's are terms, and `~>` is a placeholder for one of the following:

- `->`, i.e. `B@{E1...En}->C`. A **scalar data expression**, read as: *the application of the method B with the arguments E1...En returns the single value C.*
- `->>`, i.e. `B@{E1...En}->>C`. A **set valued data expression**, read as: *the application of the method B with the arguments E1..En returns the value C (and may also return other values that are defined through other method expressions).*
- `=>`, i.e. `B@{E1...En}=>C`. A **scalar signature expression**, read as: *there is a a scalar method whose arguments are from E1...En respectively and whose range are the instances of C.*
- `=>>`, i.e. `B@{E1...En}=>>C`. A **set valued signature expression**, read as: *there is a set valued method whose arguments are from E1...En respectively and whose range are the instances of C.*

A method expression where the list of arguments `{E1,...En}` is empty can be written as `B~>C`.

F-atoms are constructed from method expressions and auxiliary symbols. The following types are distinguished:

- Subclass assertions `A::B`, where A and B are terms. The meaning of these statements can be intuitively understood as: *A is a subclass of B.*
- Is-a assertions `A:B`, where A and B are terms. This can be read as: *A is an instance of B.*
- Object atoms `A[B@{E1...En}~>C]`, where A,B,C and all E's are terms and `~>` is a placeholder as defined above.

For relationships that are best defined in predicate syntax, F-logic includes **P-atoms**. A P-atom is a structure `p(A1...An)`, where p is a predicate symbol and `A1...An` are terms.

Out of atoms (F-atoms and P-atoms), **F-molecules are constructed**. Atomic molecules can be combined into complex molecules according to the following rules:

- Every atom is also an atomic molecule.
- Atomic object molecules `A[Me1],A[Me2],...,A[Men]`, where A is a term and all ME's are arbitrary method expressions, can be combined into a complex F-molecule `A[Me1;Me2;...;Men]`.
- An Is-a assertion `A:B` and an object molecule `B[Me1;Me2;...;Men]`, where A and B are terms and all ME's are arbitrary method expressions, can be combined into a complex F-molecule
  `A:B[Me1;Me2;...;Men]`.
- A subclass assertion `A::B` and an object molecule `B[Me1;Me2;...;Men]`, where A and B are terms and all ME's are arbitrary method expressions, can be combined into a complex F-molecule
  `A::B[Me1;Me2;...;Men]`.

If `F` is an F-molecule, then `F@A` is an **MF-molecule**, where A is a term. This can be read as *The molecule F stated in module A*. As shorthand a MF-molecule can be written as `A`, which is equivalent to `A@DefaultModule`, where `DefaultModule` is a designated term standing for the default module.

F-logic **literals** are MF-molecules that may be negated, i.e. if `A` is a MF-molecule than `A` and `NOT A` are literals.

F-formulae are created from literals by the following rules:

- All literals are F-formulae.
- if `A` and `B` are F-formulae, then so are `A OR B` and `A AND B`.
- if A is a F-Formulae and V is a variable, then `FORALL V A` and `EXISTS V E` are formulae. `FORALL` and `EXISTS` are the quantifiers for the variable V.

An F-logic **fact** is a MF-molecule that is ground. A fact is written as `A.`, where `A` is the MF-molecule.

An F-logic **rule** is a structure of the form `A <- B`, where the head A is a conjunction of MF-molecules and the body B is a F-formulae and the following conditions are met:

- All variables appearing in A or B are bound to exactly one quantifier.
- All variables appearing in A also appear in at least one positive literal in B.

By convention a rule is written as `FORALL V1,V2,...Vn A<-B`, where `V1,...Vn` are all all-quantified variables appearing in `A` and `B`. The syntax also allows to name a rule by prefixing its declaration with `Rule name:`.

A F-logic **query** is a rule without a head, i.e. `<-B`.

A F-logic **program** is a set of f-logic facts and rules.

2.2.3 Translating F-logic/LP-O Programs to Normal Logic Programs

The semantics of F-logic/LP-O is defined by a syntactic mapping between the language and (a subset of) normal logic programs. This translation is also the basis for the query evaluation in the F-logic/LP-O implementations described in later sections.

This translation consists of the following steps:

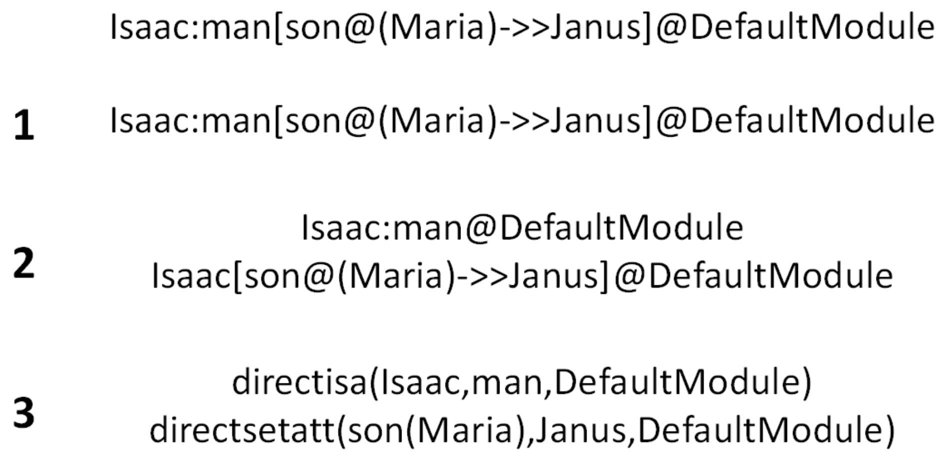
1. Simplify complex formulae with Lloyd Topor transformation.
2. Decompose complex molecules into atomic MF-molecules.
3. Translate F-logic expressions to predicate syntax.
4. Add axiom rules to the knowledge base.

Step 2 directly follows from the construction of complex molecules as detailed in the previous section. The other steps will be detailed in the following sections.

Before describing the details of the transformation, the example in figure 2.3 will give a quick overview.

The example uses the same MF-Molecule introduced earlier. The first step, the application of the Lloyd Topor Transformation does not change it, since it already has a simple structure. The second step, the decomposition into atomic molecules results in two atomic molecules. Finally the last step removes all F-logic specific syntax and creates two normal logic facts. The fourth step, the adding of the axioms, does not change these statements and is omitted.

Realizing F-logic in this way allows programs to be created using a syntax and semantics matching peoples intuition about objects, classes and



```
    Isaac:man[son@(Maria)->>Janus]@DefaultModule

1   Isaac:man[son@(Maria)->>Janus]@DefaultModule

          Isaac:man@DefaultModule
2     Isaac[son@(Maria)->>Janus]@DefaultModule

        directisa(Isaac,man,DefaultModule)
3   directsetatt(son(Maria),Janus,DefaultModule)
```

Figure 2.3: *F-logic to normal logic progran transformation example*

methods; and, at the same time, to evaluate these programs using the well understood theory of normal logic programs.

2.2.4 Lloyd Topor Transformation

Before the Lloyd-Topor transformation can be applied, it has to be ensured that all rules have a head consisting of only an atomic MF-molecule. For this reason all complex MF-molecules in rule heads are decomposed into atomic ones (according to section 2.2.2) and each rule `A1 AND A2 AND ... AND A3 <- C` (where all An's are simple MF-molecules and C is an arbitrary formulas) are replaced by n rules `A1<-C`, `A2<-C`,...,`An<-C`.

J.W. Lloyd and R. W. Topor defined a series of transformations that allow to transform a rule base with arbitrary first order logic formulas as body into a normal logic program [101, 100]. The Lloyd-Topor transformation is defined as the application of the rules detailed below until no rule is applicable anymore; [101] proves that this process always terminates and always results in a normal logic program. `H` is a simple MF-molecule, `A,B,C,V,W,X` are arbitrary formulae. Note that for this presentation `NOT` is binding stronger than `AND`.

- Replace `H <-A AND NOT (V AND W) AND B` with
  `H <-A AND NOT V AND B` and `H<-A AND NOT W AND B`.
- Replace `H <-A AND (FORALL X1...Xn W) AND B` with
  `H <-A AND NOT (EXISTS X1...Xn) NOT W AND B`.

- Replace `H <-A AND NOT (FORALL X1...Xn W) AND B` with `H <-(EXISTS X1...Xn) NOT W AND B`.
- Replace `H <-A AND V OR NOT W AND B` with `H <-A AND V AND B` and `H <-A AND NOT W AND B`.
- Replace `H <-A AND NOT (V OR NOT W) AND B` with `H <-A AND NOT V AND W AND B`.
- Replace `H <-A AND (V OR W) AND B` with `H <-A AND V AND B` and `H <-A AND W AND B`.
- Replace `H <-A AND NOT (V OR W) AND B` with `H <-A AND NOT V AND NOT W AND B`.
- Replace `H <-A AND NOT NOT W AND B` with `H <-A AND W AND B`.
- Replace `H <-A AND (EXISTS X1...Xn W) AND B` with `H <-A AND W AND B`.
- Replace `H <-A AND NOT (EXISTS X1...Xm W) AND B` with `H <- A AND NOT p(Y1...Yn) AND B` and `p(Y1...Yn)<- (EXIST X1....Xn W)` where y1,...,yk are variables in W and p is a new predicate not already appearing in the program.

Note that all variables not explicitly quantified are understood as implicitly universally quantified.

2.2.5 Predicate Syntax

In the second to last step, F-logic statements are translated into predicate syntax. This transformation is defined at the level of atomic MF-molecules as follows:

Simple FM-molecule	predicate (when contained in an F-logic rule)
`A[B@(E1...En)=>C]@D`	`atttype(A,B(E1...En),C,D)`
`A[B@(E1...En)=>>C]@D`	`setatttype(A,B(E1...En),C,D)`
`A[B@(E1...En)->C]@D`	`att(A,B(E1...En,C,D)`
`A[B@(E1...En)->>C]@D`	`setatt(A,B(E1...En,C,D)`
`A:B@C`	`isa(A,B,C)`
`A::B@C`	`sub(A,B,C)`
`p(E1...En)@B`	`p(E1...En,B)`

Directly asserted FM-molecules (i.e. facts) are translated differently, this transformation is described in the table below. Handling these translations differently enables the creation of queries such as 'what are the asserted concepts of an instance'; excluding all is-a relations that have been inferred by rules. Such queries are important e.g. for the detection of potential programming problems through anomaly detection heuristics (see section 6). The relation between the different predicate symbols used for asserted and inferred atoms is defined by six axioms described below.

Simple FM-molecule	predicate
`A[B@(E1...En)=>C]@D`	`directatttype(A,B(E1...En),C,D)`
`A[B@(E1...En)=>>C]@D`	`directsetatttype(A,B(E1...En),C,D)`
`A[B@(E1...En)->C]@D`	`directatt(A,B(E1...En),C,D)`
`A[B@(E1...En)->>C]@D`	`directsetatt(A,B(E1...En),C,D)`
`A:B@C`	`directisa(A,B,C)`
`A::B@C`	`directsub(A,B,C)`
`p(E1...En)@B`	`p(E1...En,B)`

The transformation presented here is only one of different ones supported by Ontobroker. It has been selected for presentation because it is the translation that preserves most of F-logic's power - other transformations forsake some flexibility for speed.

2.2.6 F-logic Axioms

A set of rules is added to the rule base to realize those portions of the F-logic language that are not directly supported by normal logic programs (such as anything related to inheritance). In the following the indispensable axioms are detailed[3].

Attribute inheritance rules ensure that signature expressions are inherited, e.g. that a class `men` being the subclass of `human` inherits the schema information that it has a method `child` with the range `person`.

```
RULE attributeInheritance1:
  FORALL M,C,A,R,C1,C2
    atttype(C,A,R,C1,C2)@M
    <- directatttype(C,A,R,C1,C2)@M.

RULE attributeInheritance2:
  FORALL M,Sub,Sup,A,R,C1,C2
    atttype(Sub,A,R,C1,C2)@M
    <- sub(Sub,Sup)@M and atttype(Sup,A,R,C1,C2)@M.

RULE attributeInheritance3:
```

[3]In the actual implementations a number of additional *convenience* axioms is added, that make certain operations a little simpler, e.g. to get a list of all modules used.

```
    FORALL M,C,A,R,C1,C2
      setatttype(C,A,R,C1,C2)@M
      <- directsetatttype(C,A,R,C1,C2)@M.

  RULE attributeInheritance4:
    FORALL M,Sub,Sup,A,R,C1,C2
      setatttype(Sub,A,R,C1,C2)@M
      <- sub(Sub,Sup)@M and setatttype(Sup,A,R,C1,C2)@M.
```

The **sub set relationship** axioms realize the relation between is-a and subclasses relations, e.g. that an instance of `men` is also an instance of all superclasses of `men`.

```
  RULE subSetRelationShip1:
    FORALL M,X,Y
      isa_(X,Y)@M <- directisa_(X,Y)@M.

  RULE subSetRelationShip2:
    FORALL M,El,Sub,Sup
      isa_(El,Sup)@M
      <- sub_(Sub,Sup)@M and isa_(El,Sub)@M.
```

The **subclass transitivity** rules realize the transitivity of the subclass relation, e.g. that `men` as a subclass of `human` is also a subclass of all superclasses of `human`.

```
  RULE subclassTransitivity1:
    FORALL M,X,Y
      sub_(X,Y)@M
      <- directsub_(X,Y)@M.

  RULE subclassTransitivity2:
    FORALL M,X,Y,Z
      sub_(X,Z)@M
      <- directsub_(X,Y)@M and sub_(Y,Z)@M.
```

The **asserted*** rules define the relation between the predicate syntax used to represent asserted molecules and the syntax used for inferred molecules. E.g. an inferred is-a statement is represented as `isa(A,B,C)` while an asserted one is represented as `directisa(A,B,C)` and the axiom `assertedIsa` introduced below infers the first representation from the second.

```
  RULE assertedAtttype:
    FORALL(A,B,C,D)
      attype(A,B,C,D) <- directatttype(A,B,C,D).
```

```
RULE assertedSetAtttype:
  FORALL(A,B,C,D)
    setattype(A,B,C,D) <- directsetatttype(A,B,C,D).

RULE assertedAtt:
  FORALL(A,B,C,D)
    att(A,B,C,D) <- directatt(A,B,C,D).

RULE assertedSetAtt:
  FORALL(A,B,C,D)
    setatt(A,B,C,D) <- directsetatt(A,B,C,D).

RULE assertedIsa:
  FORALL(A,B,C)
    isa(A,B,C) <- directisa(A,B,C).

RULE assertedSub:
  FORALL(A,B,C)
    sub(A,B,C) <- directsub(A,B,C).
```

Please note that none of these axioms use the signature expressions to enforce or check the signature. Any checking of type constraints is not done at the level of the language, but left to the developers of individual systems. This is discussed in detail in chapter 6.2.

2.2.7 Comparing F-logic/LP-O and F-logic/LP

In the preceding section F-logic/LP-O was presented; the version of F-logic implemented by Ontobroker that was the basis for implementations presented in later chapters. To allow the reader to put this choice into context, the following section gives a short overview of the main differences between F-logic/LP-O and F-logic/LP.

1. **Inheritable Data Expressions:** In addition to the (scalar and set valued) data expressions presented, F-logic/LP also supports inheritable data expressions. Data expressions defined in this way are inherited to all instances of the entity they are defined for.

2. **Modules:** The modules and MF-statements presented above have no correspondence in F-logic/LP.

3. **F-molecules involving is-a/subclass:** Complex F-molecules involving is-a axioms are not defined in the F-logic/LP syntax, F-logic/LP does not know modules.

4. **Equality in rule heads:** F-logic/LP allows asserting equality between objects - something that is not allowed in F-logic/LP-O or Flora [174], the other major F-logic implementation available today.

2.3 RDF and Logic Programming

RDF [29], the **R**esource **D**escription **F**ramework, is a universal data exchange language defined by the World Wide Web Consortium W3C. Its model is that of a named graph that is defined through triples consisting of a subject, a predicate and an object. Logic programming rules in conjunction with RDF are also used in parts of the implementation described in later chapters.

```
@prefix rdf:
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix zach: <http://www.fzi.de/ipe/zach/example#> .

zach:Abraham, rdf:type, zach:man.
zach:Sarah, rdf:type, zach:woman.
zach:Isaac, rdf:type, zach:man.
zach:Isaac, zach:father, zach:Abraham.
zach:Isaac, zach:mother, zach:Sarah.
```

In RDF entities are denoted by URIs. In order to still keep RDF files readable, however, URIs are usually split into the prefix and the name of the entity, and the prefix is defined only once at the beginning of the file. In the example two prefixes named `rdf` and `zach` are defined in the first three lines.

All consecutive lines then define one triple each. The first line defines that between the entity `zach:Abraham` (or actually `http://www.fzi.de/ipe/zach/example#Abraham`, after the prefix has been expanded) and the entity `zach:Man` there is a relation denoted by `rdf:type`. The remaining lines then define `Isaac` to be of type `man` and `Sarah` to be of type `woman`. It further states relations between these entities, namely that `Abraham` and `Sarah` are father and mother of `Isaac`.

The model created by these statements is commonly shown as a graph as shown in figure 2.4.

These RDF statements can also be understood as facts in a normal logic program and rules can be used to infer additional information from them. Consider the following rule as an example:

```
@prefix rdf:
```

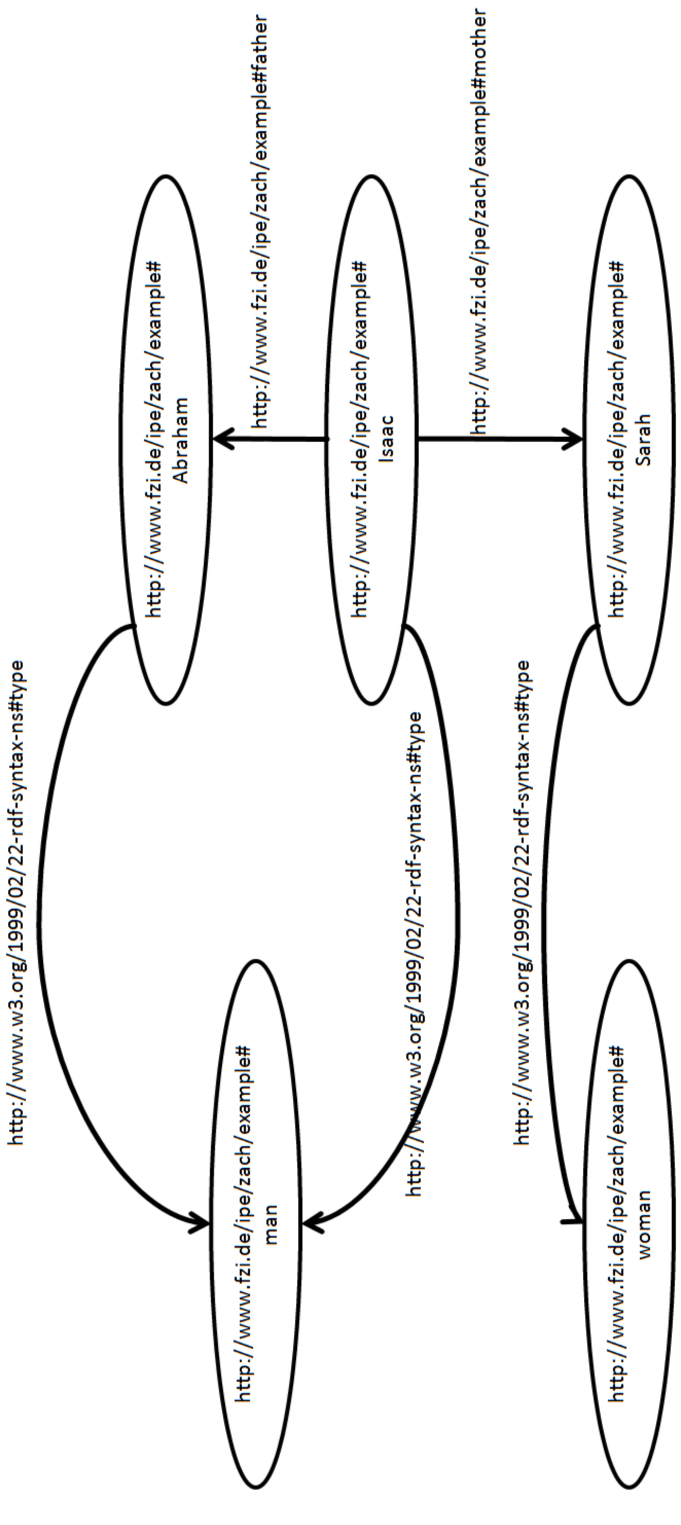

http://www.fzi.de/ipe/zach/example#father
http://www.fzi.de/ipe/zach/example#mother
http://www.fzi.de/ipe/zach/example# Abraham
http://www.fzi.de/ipe/zach/example# Isaac
http://www.fzi.de/ipe/zach/example# Sarah
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.fzi.de/ipe/zach/example# man
http://www.fzi.de/ipe/zach/example# woman

Figure 2.4: *Example RDF Graph*

```
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
   @prefix zach:
      <http://www.fzi.de/ipe/zach/example#> .

   [ruleExample:
      (?x zach:son ?y) <-
         (?x rdf:type zach:man)(?y zach:father ?x)]
```

After the already familiar prefix declarations the rule starts with the declaration of a name, `ruleExample` in this case. The next line is the head of the rule; it defines the kind of RDF triples deduced by this rule. In this example rule these triples have the form `(?x zach:son ?y)`, were `?x` and `?y` are variables, identified by their starting `?`. The third line of the rule defines the rule body. The body in the example consists of two triple patterns implicitly connected through an conjunction. Altogether the rule can be read as: x is the son of y, if x is of type man and y is the father of x.

Logic Programming rules on top of RDF are also used in some of the implementations presented in later chapters and are hence detailed in the sections below.

2.3.1 The RDF Abstract Syntax

The semantics of RDF is defined over its abstract syntax [29].

An **RDF graph** is a set of **RDF triples**. Each RDF triple has three parts:

- A **subject**, which is a URI reference or a blank node.
- A **predicate**, which is a URI reference.
- An **object**, which is a URI reference, a blank node or an RDF literal.

A RDF triple is commonly written as `subject,predicate,object`. The **nodes** in an RDF graph are the terms used as subject or object in a triple of that graph.

An example triple could be the following:

```
http://www.fzi.de/ipe/zach/example#Abraham,
http://www.fzi.de/ipe/zach/example#age,
175
```

Intuitively it can be said that this triple defines the age of Abraham to be 175. The subject of this triple is an URI representing the biblical Abraham (unlike the string "Abraham", which is the first name for many thousand persons). The predicate is the uri `http://www.fzi.de/ipe/uach/example#age` and the object is the literal 175.

A **URI reference** is a Unicode string that can be translated into a valid absolute URI [14] by encoding it as UTF-8 and escaping symbols not corresponding to valid US-ASCII symbols.

RDF literals are further discriminated into **typed literals** and **plain literals**. Plain literals consist of a lexical form and an optional language identifier [1] in lower case. Typed literals consist of a lexical form and a datatype URI which is a URI reference. The lexical form of an RDF literal is a Unicode string.

Examples for plain literals are `"Esau"` or `"Kindergarten"@en`; the first only denoting the string itself and the second also indicating that the string is a text in the language `en`; an acronym for English as defined in [1]. An example for a typed literal is the following:

`"2002-10-10Z"^^http://www.w3.org/2001/XMLSchema#date.`

It defines the string "2002-10-10" and stipulates that this string needs to be interpreted according to the rules associated with the datatype URI `http://www.w3.org/2001/XMLSchema#date`. These rules are defined in [17] and for this concrete URI specify that this string should be understood as the duration starting on the 10th of october 2002 at 0:00 coordinated universal time (UTC) and ending at, but not including 24:00 of the same day.

Blank nodes are elements from an infinite set, disjoint with both literals and URIs. Blank nodes can be intuitively understood as existentially qualified variables.

`_1 #father #Abraham.` is an example for a RDF triple involving a blank node, where `_1` denotes the blank note and `#father` and `#Abraham` are URIs that have been shortened for readability. Intuitively this triple can be read as: there is some (unindentified) father of Abraham.

2.3.2 Turtle - Concrete RDF Syntax

Actual RDF data can be created using one of a number of different concrete syntaxes [10, 15, 11, 73]. Throughout this thesis the Turtle syntax [11, 9] will be used for its simplicity, conciseness, readability and its focus on pure RDF. The complete specification of Turtle is given in [9], only a short overview is given here.

The basic building block of the Turtle syntax are triples, with subject, predicate and object enclosed in <, > and separated by whitespaces; the triple ending with a dot. For example:

```
<http://fzi.de/subject> <http://fzi.de/predicate>
   <http://fzi.de/object>.
```

Prefixes can be defined with the `@prefix` element to shorten the URIs. For example the following statements assert the same triple as example above, but are better readable.

```
@prefix fzi: <http://fzi.de/>.
   fzi:subject fzi:predicate fzi:object.
```

A default prefix can be defined by using `@prefix :`, e.g.

```
@prefix : <http://fzi.de/>.
   :subject :predicate :object.
```

Multiple statements sharing the same subject and predicate can be combined using commas, e.g. (prefix declaration omitted for conciseness)

```
   :someSubject :somePredicate :someObject,
                               :anotherObject,
                               :evenMore.
```

Statements sharing only the subject can be combined using semicolons, e.g.

```
   :someSubject :somePredicate :someObject;
                :anotherPredicate :anotherObject;
                :oneMorePredicate :oneMoreObject.
```

Blank nodes can be introduced using the `_:` prefix, e.g.

```
 :someSubject :somePredicate _:blankNode.
 _:blankNode :someRelation :otherObject.
```

The triples above can be defined equivalently using the `[...]` syntax that can be used to introduce blank nodes only needed once:

```
 :someSubject :somePredicate [
      :someRelation :otherObject ].
```

Typed literals are written in double quotes followed by `^^` and the datatype URI (that can be appreviated using prefixes), e.g.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
:someSubject :somePredicate "12"^^xsd:integer.
```

Literals with a language identifier are written in double quotes followed by `@` and the language tag, e.g.

```
:someSubject :somePredicate "Some string"@en.
```

2.3.3 The Semantics of RDF

The semantics of RDF [83] can be understood as consisting of two parts: those directly associated with the abstract syntax and the semantics of the RDF vocabulary. Only the first part is necessary for understanding the content of the thesis and hence only this part is presented. The RDF vocabulary defines a number of reserved URIs such as type or list and their meaning. Readers interested in the vocabulary may consult [83].

In this section the semantics of RDF is presented on the basis of Herbrand models - as described by [43, 48]. Doing so makes it easier to see the correspondence to normal logic programs.

Using the syntax of normal logic program as defined in section 2.1.1, we can write a triple s, p, o as $triple(s, p, o)$, where s and o are terms and p is a ground term. A RDF graph then is a set of statements of this form. A Herbrand interpretation of this graph (as defined in section 2.1.3) is a model, iff all ground triples are in the interpretation and for each non-ground triple t there exist a ground substitution θ such that $t\theta$ is in the interpretation.

To define the semantics we first obtain the **canonical graph** from an RDF graph by

1. Completing it by adding the RDF axiomatic triples ([83] - section 3.1) and applying the RDF entailment rules ([83] - section 7.2).
2. Replacing all typed literal values by their canonical representation as defined in [17].
3. Removing all blank nodes through Skolemization; i.e. blank nodes are replaced by ground terms not used otherwise.

The Herbrand model (as defined in section 2.1.3) of this canonical graph is called the canonical model. According to [48] these canonical models contain explicitly all information entailed by the normative RDF semantics defined by [83] in the following way: An RDF graph R entails a graph E, $R \models E$ iff one model of E is a sub graph of the canonical model of R. Because of this property this model can be used to define the results of queries to the RDF graph - in the RDF query language SPARQL [48] or in normal logic programs as described below.

Note that once the RDF graph is transformed into the canonical graph, the Herbrand model is defined exactly as it is for normal logic programs. Hence an RDF graph can be translated into ground facts for a normal logic program.

2.3.4 RDF Rules

Once the semantics of RDF is defined in a framework compatible with normal logic programs, the data can be further processed and queried using the rules and queries defined in the beginning of this chapter. Doing so gives a versatile tool and well understood framework for processing and using this data. This has been done in the well known Jena semantic web framework [30] that is also the basis of some of the implementations presented later. The Jena framework only implements a subset of normal logic programs as presented earlier: it does not allow for function symbols[4] nor negation. A Jena-RDF rule consists of a head and a body, each consisting of a set of triples. These rules are commonly written as:

```
@prefix fzi: <http://fzi.de/#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

[ruleExample:
   (?x fzi:son ?y)
  <-
   (?y rdf:type fzi:man),
   (?y fzi:mother ?x)]
```

This rule is stating that: if y is of type man and y has a mother x, then x has a son y. Using the triple predicate (and omitting the URI prefixes for brevity) this can be written as a normal logic rule:

$$triple(X, son, Y) \leftarrow triple(Y, type, man), triple(Y, mother, X)$$

An RDF rule set and an RDF graph can then be seen as a normal logic program consisting of a set of ground facts and a set of rules. The ground facts are obtained by a syntactic translation of the triples defining the RDF graph as defined in section 2.3.3. The rules are constructed by the rule translation at the beginning of this section. The resulting normal logic program is restricted in that it does only contains ground facts, only one predicate (triple) and no function symbols. The semantics of this RDF rule set is defined by the semantics of the normal logic program that results from this translation.

[4]The *functor* defined in the Jena rules syntax is only syntactic sugar allowing easier access to data structures consisting of multiple triples, it is not a function symbol in the sense of logic programming.

Chapter 3

Analysis and Design

It is evident that no computer program of meaningful size can be constructed without faults. This is not due to sloppiness or lack of trying by the programmers but because a program is a collection of arbitrarily complex assumptions whose ultimate consequence cannot, in any finite time, be foreseen [151]. It is for this reason that the correctness of a program cannot - in general - be proven and that almost no software system is without faults. Software quality assurance (SQA) - the process to ensure that a program conforms to its specification - therefore centers to a great extent on reducing the number of faults in a program.

Rule based systems are often presented as a simpler and more natural way to build computer systems - compared to both imperative programming and other logic formalisms such as full first order logic or desciption logics. It seems natural to assume that this simplicity also applies to finding and preventing faults; to SQA in general. However, with respect to debugging of rule bases this promise of simplicity remains elusive. Indeed a survey found (see section 3.1.1) that most developers of rule based systems think that the debugging of rule based systems is more difficult than the debugging of 'conventional' - object oriented and procedural programs. An observation that the author also found corroborated in three rule base development projects he participated in.

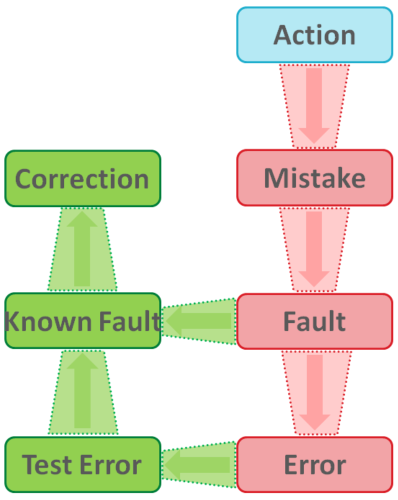


Figure 3.1: *The Fault Lifecycle*

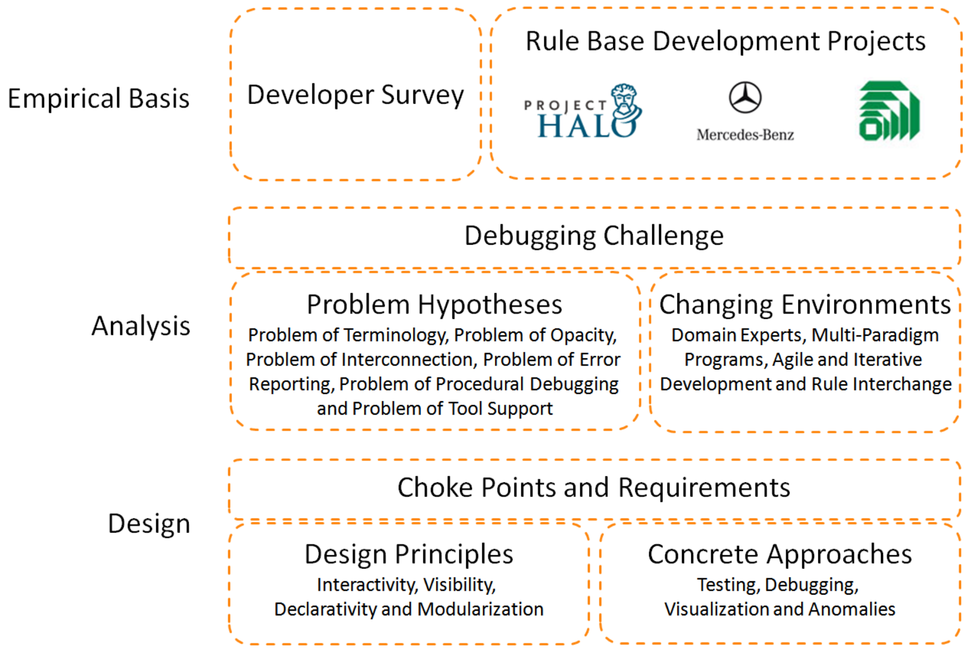

Empirical Basis
Developer Survey
Rule Base Development Projects
PROJECT HALO
Mercedes-Benz
Analysis
Debugging Challenge
Problem Hypotheses
Problem of Terminology, Problem of Opacity, Problem of Interconnection, Problem of Error Reporting, Problem of Procedural Debugging and Problem of Tool Support
Changing Environments
Domain Experts, Multi-Paradigm Programs, Agile and Iterative Development and Rule Interchange
Design
Choke Points and Requirements
Design Principles
Interactivity, Visibility, Declarativity and Modularization
Concrete Approaches
Testing, Debugging, Visualization and Anomalies

Figure 3.2: *Overview of the analysis and design chapter*

The goal of this chapter then, is to analyse this debugging challenge; to generate hypotheses for its explanation and to derive a design for tool support that can help tackle it.

Towards this end this chapter takes a broad view on the debugging challenge and considers the entire fault lifecycle (see figure 3.1). The fault lifecycle is a conceptual model created in this thesis to order and prioritize issues affecting the quality assurance in rule bases. In this chapter the fault lifecycle is used to order and aggregate the results from the analysis at the beginning of this chapter. The fault lifecycle starts with a developer doing a mistake which may result in a fault in the rule base. This fault may then later manifest itself in a computation, resulting in an error. Testing and static analysis techniques can be used to identify these faults either based on a error observed in testing or directly in the program code. The fault lifecycle is presented in more detail in section 3.2.2.

The overall structure of this chapter is given in figure 3.2. This chapter is structured in three main sections: the introduction of the empirical basis for the analysis, the analysis itself and finally an overall design. The very coarse design at the end of this section is then instantiated in the later chapters of this thesis.

The empirical basis consists of a developer survey (section 3.1.1) and mostly qualitative evidence and experience from three rule base development projects. The largest of these, Project Halo, is described in section 3.1.2, the two smaller ones jointly in section 3.1.3.

The Analysis starts with the evidence for the existence of the debugging challenge (section 3.2.1) followed by a more thorough elaboration on the fault lifecycle (section 3.2.2) that guides the rest of the analysis. The core of the analysis then consists of six hypotheses that can explain (part of) the debugging challenge (section 3.2.3). In addition, four properties that characterize modern rule base development and also affect quality assurance are presented (section 3.2.4).

In the design section the results from the analysis are aggregated to identify choke points and requirements (section 3.3.1); to identify those parts of the fault lifecycle that should be better supported with tools. Based on this, four concrete approaches are sketched (section 3.3.3), each of which then is the subject of a later chapter in this thesis. Design principles - orthogonal to the actual tools - are also presented to address further issues from the analysis (section 3.3.2).

3.1 Empirical Basis

This section introduces the empirical basis for the analysis of the debugging challenge. The empirical basis consists of the results from a survey of developers of rule based systems and of experiences from three rule base development projects.

This section starts with a description of the results from the developer survey. After that Project Halo is described as the largest of the development projects followed by two smaller projects (F-Verify and Online Forecast) that are jointly described in the last part of this section.

3.1.1 Developer Survey

Within literature there is little empirical data and overview about the way rule bases are used, developed and which challenges developers of these systems face; in sum there is little data that could be used to set priorities for research and development. To address this shortcoming I conducted a survey of the methods and tools used and the issues facing the development of rule based systems. Based on the results from older surveys [76, 123] verification and debugging issues had been identified as particular important and I designed the survey to focus on these.

This section starts with a short related work section introducing two much older studies that addressed similar questions: results from these surveys are also cited throughout the text were similar questions were used. The next sections introduce the survey and present data about the participants, their experience and the rule based systems they develop. The core results from the survey are then grouped into three sections, (1) Methods and Tools Used for Development, (2) Verification, Bugs and Debugging and (3) Issues and Comparison to Procedural Programming.

The goal of this section is to introduce the survey and its results in its entirety. The conclusions from the survey and their influence on the design of the verification architecture are discussed in the analysis section (section 3.2).

Related Surveys

In 1991, Hamilton et al. [76] questioned 70 people with respect to the state of the practice in knowledge-based system verification and validation. The goal of this survey was to document the state of the practice. The insight gathered during the survey and follow up interviews was used to develop recommendations for further V&V research. The findings from this survey included that 62% of developers judged the developed expert system to be less accurate than the expert and 75% judged it to be less accurate than expected.

Also published in 1991, O'Leary [123] used a mailed survey to query 34 developers of knowledge-based systems. This poll had the specific goal of testing a number of hypotheses with respect to prototyping of knowledge based systems. The core finding was that the prototyping development methodology is found to lead to more robust models and that it does not increase the validation difficulty.

In the broader world of *conventional* software engineering (i.e. using procedural or object-oriented languages) Cusumano et al. [42] compared the practices used in software development across countries, finding in particular that Indian software development projects used the most elaborate practices, combining traditional techniques such as specification and reviews with modern ideas like pair programming. Zhao and Elbaum [178, 179] explored the use of quality assurance tools and techniques in open source projects. The most interesting findings from these studies include, that while over half of the projects spend more than 20% of the development time on testing, only 5% compute any test coverage measures and the use of regression testing is not widespread. Finally, Runeson et al. [142] summarize the (mostly experimental) empirical data with respect to quality assurance methods. Among other things they found that on average only 25% to 50% of faults are found during inspection and only a slightly higher 30% to 50% during testing - concluding that on average half of the faults remain.

Survey Construction

The goal of the survey was to be an exploratory study of the methods and tools used, and the issues facing the developers of rule based systems. The survey focused on verification and in particular debugging as very important questions that in the author's experience are particularly problematic for the development of rule based systems. Some questions were also derived from specific hypotheses, described in detail below together with the results for the questions.

The survey was designed to be answerable in less than 15 minutes; included 17 questions spread over three pages and was conducted using the SurveyMonkey [157] service. The survey with all questions is included in the appendix A. Participants were asked to answer all questions with respect to the largest rule base in whose development they had been involved

	Mean	Median	Std. Deviation
Person Month Development			
PM for entire software	59	15	148
PM for rule base	9	5.5	15
Size of Rule Base			
Number of rules	1969	120	8693
Size of average rule	9.3	5	17
Size of largest rule	24	11	39
Developers involved			
Rule developers	3	2	4
Other software developers	3	1	8
Domain experts that created rules	1.5	1	2
Domain experts as consultants	1.9	1	2.5
Domain experts for V&V	1.7	1	2.4
Others	0.6	0	1.6

Table 3.1: *Measures of the size of the rule base*

with in the past 5 years.

Participants

Participants were recruited through emails sent to public mailing lists concerned with rule based systems[1], Jess and mailing lists of academic institutes; invitations were also published on some blogs concerned with rule based systems[2] . A chance to win a camera was given as additional incentive to motivate people to participate. 76 people opened the survey and 64 answered most questions; one reply was removed because it consisted of obviously nonsensical answers.

For the purpose of analysis the wide variety of systems used by the respon-

[1]The mailing lists where the CLIPS mailing list, RuleML 'all' mailing list, the RIF initiative mailing list, the JESS users mailing list, the GNU Prolog users mailing list, the SWI Prolog mailing list, the TU Prolog users mailing list, the Pellet users mailing list, the XSB users mailing list, the Jena developer mailing list, the Drools developer mailing list and the Semantic Web mailing list of the W3C

[2]Smart Enough System at http://smartenoughsystems.com/wp/, James Taylors Decision Management http://www.ebizq.net/blogs/decision_management/, Enterprise Decision Management Blog http://www.edmblog.com/ and finally the author's own blog at http://vzach.de/blog

dents was grouped into five groups:

- **Prolog:** 7 results; consisting of tuProlog(2), SWI Prolog (2), Visual Prolog (1), XSB (1) and Yap Prolog (1)
- **Declarative Rules:** 11 results; consisting of F-logic - ontoprise (3), SWRL - KAON2 (2), SWRL - Protege SWRL Tab (2), SWRL - PELLET (1), Jena Rules (1), WSML-Flight (1), Ontology Works Remote Knowledge Server (1) and Iris (1)
- **Business Rule Management Systems (BRMS):** 17 results; consisting of JBoss /Drools (8), Fair Isaac Blaze Advisor (3), Yasu Quick Rules (SAP) (2), BizTalk (1), NxBre (1), Acumen Business-Rule Manager(1), OpenRules (1) and Ilog JRules/.Net rules (1)
- **Shells:** 24 results, consisting of Jess (12), Clips (9), Mandarax (1), Jamocha (1) and KnowledgeWorks/LispWorks (1)
- **Other:** 1 results, using a 'Proprietary IT software engine'

The size of the reported systems varied widely (see Table 3.1); the average rule base consists of 2000 rules, has 9 conditions/body atoms per average rule and is developed in 9 person months. However, the average size is strongly influenced by a small number of very large systems, half of the systems have not more than 120 rules. On average the rule base is part of a much larger software system that takes almost 60 person months to develop. The largest system in the survey has 63000 (partly learned) rules and is used for disease event analysis. The most time consuming took 100 person months to develop and is used to determine parameters to operate a medical imaging system. Slightly over 50% of the projects involve at least one domain expert that creates rules herself.

On average the people filling out the survey had 6.6 years of experience with rule based systems and 15 years experience with creating computer programs in general.

The tasks of the rule bases (entered as free text) include workflow management, diagnosis, scoring, ontology reasoning, tutoring and planning. The rule bases are created for a multitude of different domains, including insurance, finance, health care, biology, computer games, travel and software engineering.

38% of the rule bases are commercially deployed, 26% are deployed in a research setting and 10% are under development with deployment planned. The remaining 26% are prototypes, 10% are prototypes that are also used by others than the developers.

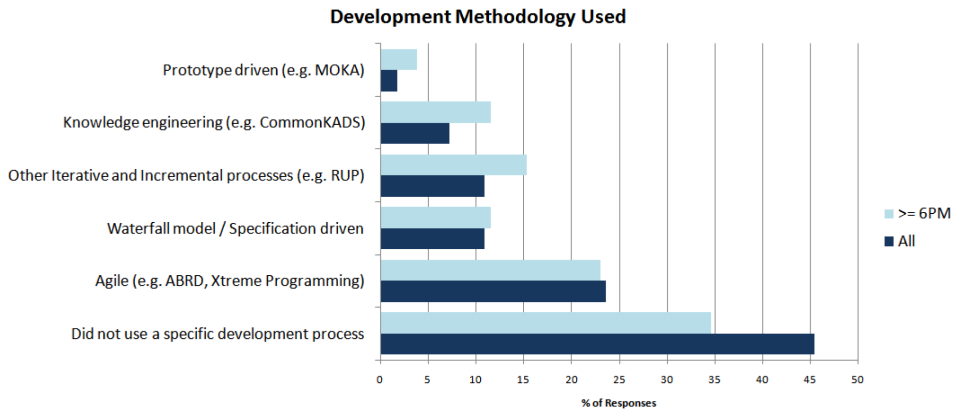


Figure 3.3: *The development methodology used, for all responses and for the 26 responses where the rule base development took at least 6 person months.*

Methods and Tools Used for Development

Participants of the survey where given a multiple choice question to describe the methods used for the development of the rule base. The answers (see figure 3.3) showed that indeed a large part of rule based systems are created without any specific development process and that the rise of agile and iterative methods [97, 102] is also visible for rule based systems. In 1991, Hamilton et al. [76] used a similar question and found that the most used model was the cyclic model (41%) and that 22% of the respondents followed no model[3].

The next questions asked participants for the tools used for the development. The results show that almost half of the respondants use an integrated development environment (IDE)[4] [124], . For editing rules the most widely used tools are still textual editors, with 33% and 28% of respondents stating that they use a simple text editor or a textual rule editor with syntax highlighting. With 26% of respondents using them, graphical rule editors are also widespread (see table 3.2).

The results show that the overwhelming majority of rule bases is still created manually; is not learned or generated as some expect. Further it shows that text editors - even simple ones without syntax highlighting - are still used, meaning that even simple typos that could be prevented by more elaborate tools will still happen in practice. Finally the questions about

[3]However, care should be taken when comparing these numbers, since the sample of the surveys differs considerable: on average the projects discussed [76] were considerable larger.

[4]Such as the Ontoprise's Ontostudio [124], Ilog Rule Studio [85], the Visual Prolog IDE [163] or the SWRL tab of Protege [70].

	% responses
Simple text editor	33%
Textual rule editor	28%
Constraint language, business language rule editor	10%
Graphical rule editor	26%
Spreadsheets based rule editor	12%
Decision trees rule editor	9%
Rule Learning	5%
An IDE that allows to edit, load, debug and run rules	46%

Table 3.2: *Tools used for rule development*

methods used revealed a preference for agile processes and even no development process at all - meaning in particular that tools cannot rely on the availability of formal specifications.

Verification, Bugs and Debugging

To gain an overview of the verification state of practice, the survey included a multiple choice question that asked participants to check all V&V tools or methods that they use in their project.

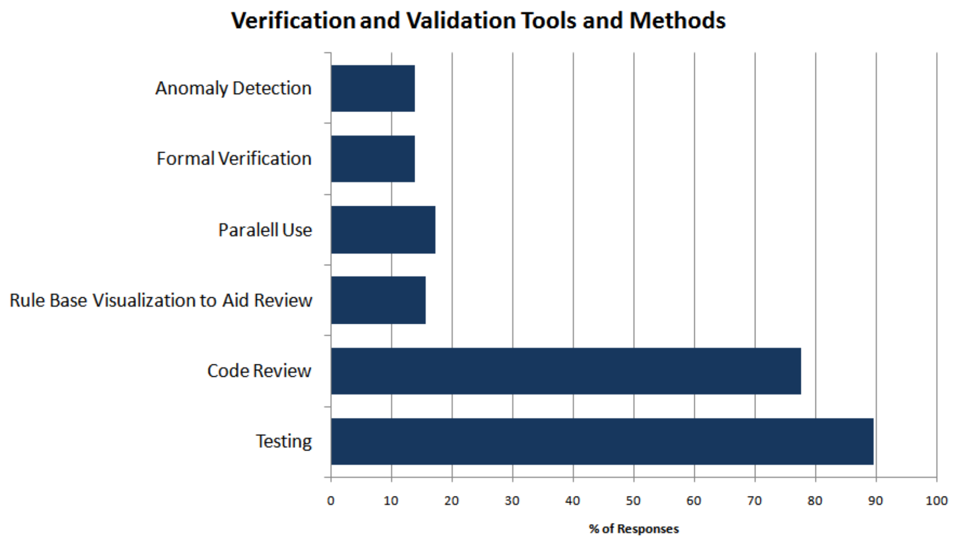


Figure 3.4: *The verification and validation methods and tools used, in percent of respondants*

The results show (a summarize is shown in figure 3.4) that verification is dominated by testing (used by 90%) and code review (used by 78%). 74% of respondents do testing with actual data, 50% test with contrived data. Advanced methods of test organization are used by a minority, with only 31% doing regression testing and 19% doing structural testing with test coverage metrics. Code review is done equally by domain experts (53%) and developers (57%), most projects combine both (73% of projects that perform a code review, have both domain experts and developers do it). The system is used parallel to development in 17% of the projects; in 16% it is used by developers; in 14% by domain experts.

O'Leary [123] posed a similar question to the developers of expert systems in 1991, asking about the validation effort spent on particular methods. In the average over all responses, he found that most effort is spent on testing with actual data (31% of validation effort), followed by testing with contrived data (17.9%), code review by domain expert (17.6%), code review by developer (13%), parallel use by expert (12%) and parallel use of system by non-expert (7%).

Debugging Tools

Debugging is dominated by procedural debuggers, i.e. debuggers similar to the ones used in procedural programming; tools that allow to specify breakpoints, to suspend the inference engine and to explore the stepwise execution of the program[5]. 37% of the projects used a command line procedural debugger and 46% a graphical procedural debugger. Explanations are used by almost a quarter (23%) of the respondents for debugging. An overview of these results are shown in figure 3.5.

Surprisingly widespread is the use of Algorithmic Debugging [151] and Why-Not Explanations (e.g. [32, 8]), considering that to the best knowledge of the author there is no widely available and currently maintained implementation of either of these debugging paradigms. For the systems used by three of the five people that professed to use Algorithmic Debugging (JBoss rules/Drools and Fair Isaac Blaze Advisor) no mentioning of any such debugger could be found and it seems likely that the short explanation for this debugging paradigm given in the survey (*'system tries to identify error by asking user for results of sub computations'*) was insufficient to convey the meaning of this concept. Similarly for Why-Not explanations three of the four respondents use systems for which such no mentioning of such debuggers could be found. The remaining two responses for Algorithmic Debugging and the remaining one for Why-Not explanations use Prolog dialects, where such debuggers have existed/exist.

[5]An overview and description of the different debugging paradigms for rule based systems can be found in [177].

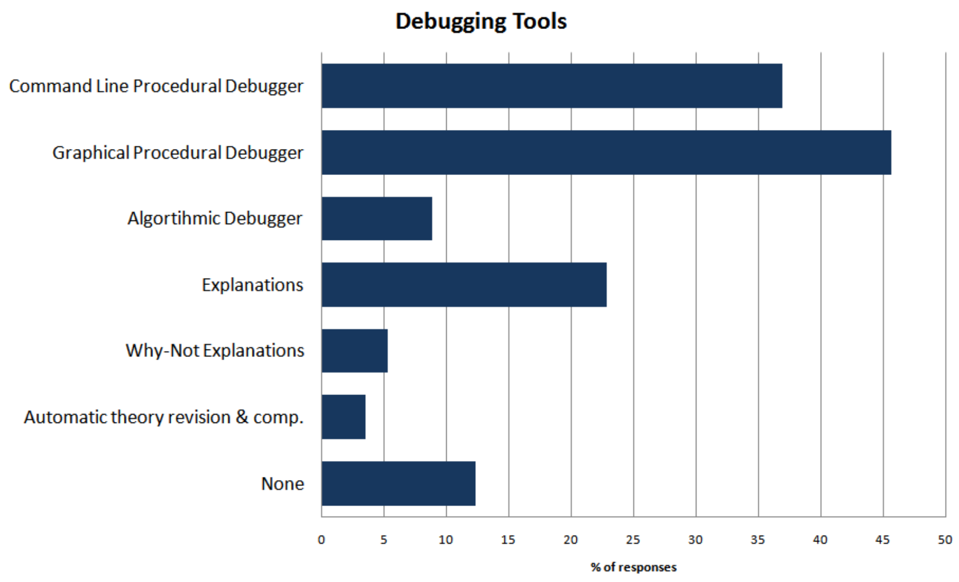


Figure 3.5: *Tools used for debugging*

Bugs, Symptoms of Faults in the Rule Base

Debugging is the process of tracking down a fault based on error revealing values. The difficulty of this process and the kind of tools that can support it, depends on the error revealing values and how well these allow for the identification of the fault. In the author's experience based on F-logic, most faults in rule based systems cause a query to not return any result (the so called *no-result-case*) or to return fewer results than expected. This stands in contrast to the development with modern object-oriented languages where in many cases at least a stack trace is available. This is problematic because a *no-result-case* gives only very little information for fault localization and means that most explanation approaches are not suitable for debugging (because these rely on the existence of a result). A question was included in the survey about the common symptoms of faults in the rule base to check whether this patterns of error revealing values holds for rule based systems in general.

The results (see table 3.3) show 'wrong results' as the most frequent bug, followed by 'no results' and 'partial results'. Most participants encounter not terminating tests and crashing rule engines only seldom. The results show also that 60% of participants frequently and 34% sometimes encounter a fault showing itself in the failure of the rule base to conclude an expected result/result part. For the developers of the system using *declarative rules* (see section 4), the no-result case is the most frequent bug. These

	frequent	seldom	never
A query/test would not terminate	7	57	30
A query/test did not return any result	38	47	11
A wrong result was returned	53	39	5
A part of the result missing	31	42	20
The rule engine crashed	9	47	38

Table 3.3: *Bugs, symptoms of faults in the rule base in percent of responses. To 100% missing percent: respondents selected* not applicable

	Avg.	Not an Issue	Annoyance	Hindrance
Debugging	1	12	28	12
Determining completeness	0.76	18	27	6
Supporting tools missing/immature	0.67	26	17	9
Editing of rules	0.66	24	23	6
Determining test coverage	0.65	25	19	7
Inexperienced developers	0.58	31	13	9
Rule expressivity	0.5	33	12	7
Keeping rule base up to date	0.5	30	19	4
Understanding the rule base	0.47	31	19	3
Runtime performance	0.41	35	14	4
Organizing collaboration	0.41	35	14	4

Table 3.4: *Issues hindering the development of rule based systems. Numbers show the actual number of participants that selected an option (except for the second column that shows the overall average, computed as described in the text).*

results underline the need for debugging approaches to support users in diagnosing bugs based on missing conclusions.

Issues and Comparison to Procedural Programming

On the last page of the survey participants were asked to rank a number of possible issues as *Not an Issue*, *Annoyance* or *Hindered Development*. An average score was obtained for the issues by multiplying the *annoyance* answers with 1, the *hindrance* answers with two and dividing by the number of all answers. The aggregated answers for this question are shown in the table 3.4[6].

[6]Please note that the *Rule Expressivity* option was phrased in a way that asked also for things that could not easily be represented, not only things that could not be represented at all.

The results show the issues of debugging, validation and tool support as the most important ones. The issues of probably the largest academic interest - runtime performance and rule expressivity, are seen as lesser problem. This is particulary interesting in the light of the fact that of the 7 survey participants that stated they were hindered by rule expressivity, none used a declarative rule system (for which these questions are debated the loudest).

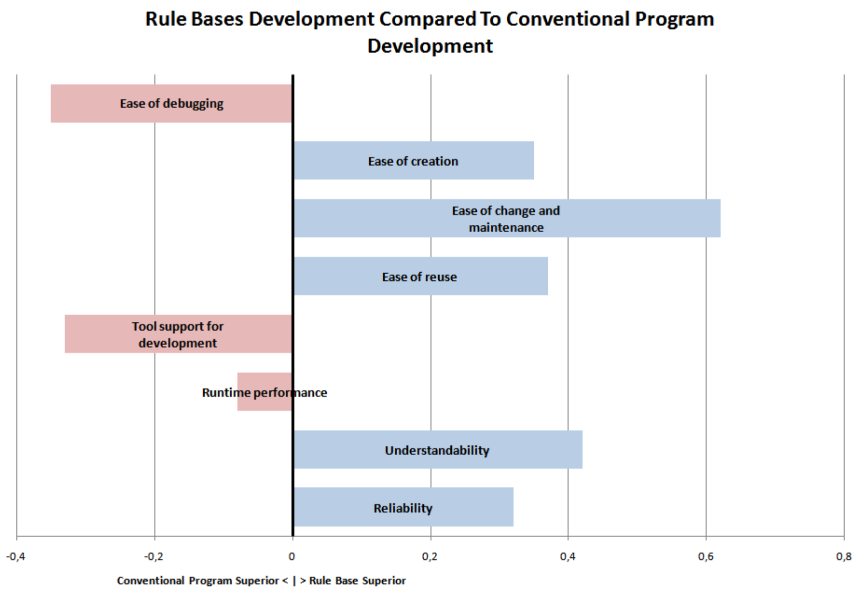


Figure 3.6: *Participants' opinion about the strengths and weaknesses of rule base development compared to that of 'conventional' programs. Positive numbers indicate that the majority thought rule bases to be superiour, negative numbers that they thought conventional programs to be superior.*

These findings of verification and validation issues as the most important ones are similar to the finding of Leary [123]. He found that the *potentially biggest problems* were determining the completeness of the knowledge base and the difficulty to ensure that the knowledge in the system is correct.

In a final question participants were asked how a rule base development process compares to the development of a conventional program (created with procedural or object oriented languages) of similar size. A number of properties was given to be ranked with *Rule base superior*, *Comparable*, *Conventional program superior* and *Don't know*. The aggregated score for each property was determined by subtracting the the number of *conventional program superior* answers from the *rule base superior* answers and dividing the result by the number of answers other than *don't know*.

The participants of the survey judged rule bases to be superior in most respects. The largest consensus was that rule bases are indeed easier to change and to maintain. Easy of creation, ease of reuse, understandability and reliability are also seen as the strong points of rule based systems. A small majority saw conventional programs as superior in runtime performance; most saw rule bases as inferior in ease of debugging support and tool support for development.

3.1.2 Project Halo

This section describes the second phase of Project Halo, another important part of the empirical basis for the analysis. The author participated almost full time in the Halo project for more than 18 months. The requirements from Project Halo inspired a large part of the approach of this thesis, a considerable fraction of the implementations and some evaluations from this thesis were done within project halo.

This section begins with a short summary of the overall goal and vision of Project Halo. Next the scope of the second phase of this project is introduced; this phase is the only one relevant for this thesis. This is followed by a description of the implementation created there: Dark Matter Studio. The remainder of this section is then dedicated to describing the evaluation of Dark Matter Studio and its results.

Vision and Earlier Phases

The long term goal of *Project Halo*[7] is to build a *Digital Aristotle*, a computer system that stores a significant part of humankind's scientific knowledge and that is able to answer novel questions (i.e. questions not known or foreseen during the creation of the system) about these parts of science. This system should be able to act both as an interactive tutor for students as well as a research tool for scientists. Project Halo is structured as a multi-stage effort, the second stage of which forms the empirical basis for parts of this dissertation.

The main goal of the first phase of Project Halo [62, 61] was to access the current state of the art in applied knowledge representation and reasoning (KR&R) systems; to establish whether existing KR&R systems could form the foundation for a Digital Aristotle. The domain chosen for this experiment was a subset of the questions from the introductory college-level Advanced Placement (AP) test[8] test for chemistry. This phase showed that it is

[7]Project Halo is funded by Vulcan Inc.

[8]The Advanced Placement Tests can be taken by college students in the USA, normally they are taken after a special course preparing for the test. Good grades on this test count as credits on some universtites and some schools also allow AP test scores to override school

indeed possible with current KR&R technology to match the performance of students on the Advanced Placement test, however, it also showed that the creation of the knowledge needed for this task is very expensive [62, 61].

Project Halo, Second Phase

Of the overall (and by the time of this writing, still running) Project Halo only the second phase is relevant for this thesis. The goal of this phase was to create and evaluate tools that allow domain experts to create the knowledge base necessary for a Digital Aristotle with ever decreasing reliance on knowledge engineers [34, 71, 39, 35]. It was hoped that this could dramatically decrease the cost of building a scientific question answering system [71] and further reduce the number of errors due to incomplete domain knowledge by the knowledge engineers. The domain chosen for the second phase was that of introductory college-level Advanced Placements tests for chemistry, biology and physics.

The second phase was conducted as a 22 month effort undertaken in two stages. The first 6 month stage was dedicated to a careful analysis of sample questions and the design of the application. This phase was followed by a 15 month implementation phase that ended with an evaluation. Three teams participated in the first stage, only two in the second stage. The project structure is such, that the different teams have the same requirements, realize these separately and are the judged on their relative performance. The results reported here are solely from team Ontoprise, which included participants from Ontoprise, Carnegie Mellon University, Open University, Georgia Tech and DFKI.

Dark Matter Studio

During the second phase of Project Halo, team Ontoprise built the *Dark Matter Studio* (DMS, an example screenshot is shown in figure **??**) tool in order to support domain experts in the creation of a scientific knowledge base. Only a short overview of this system will be given here.

Dark Matter Studio is built to support a *document rooted* methodology [62] of knowledge formulation. This methodology stipulates that domain experts use an existing document (such as a textbook) as basis for formulating knowledge. Having a root document as a foundation should help the experts to decide what to model and in which order. It is further assumed that textbooks are carefully structured in a way that is well suited

grades.

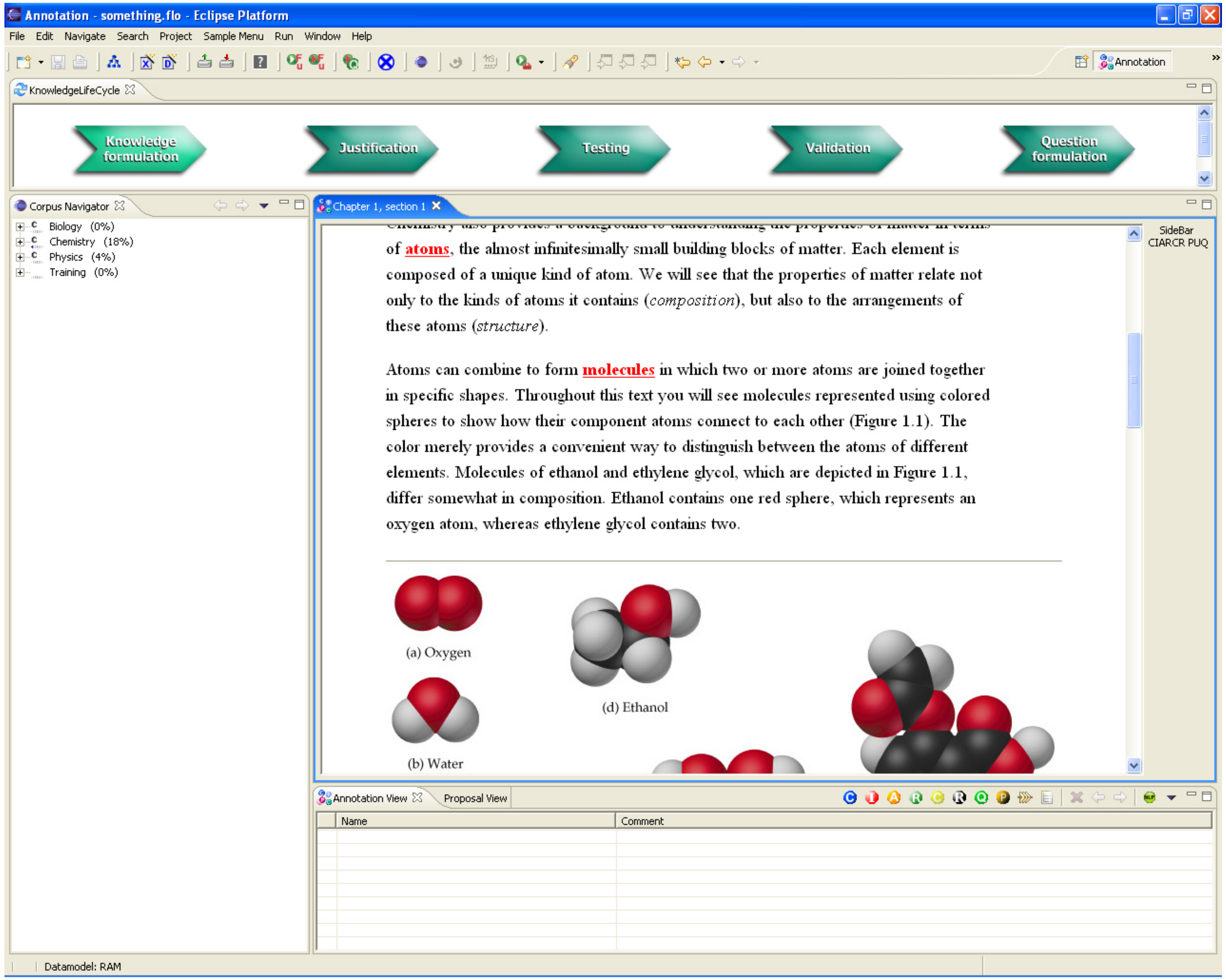
for knowledge formulation. All created knowledge is tied to the document; the document can thus help to contextualize and explain the contents of the knowledge base.

Dark Matter Studio is created in a way that isolates the user from the details of the knowledge representation language and the workings of the inference engine. The concepts the user manipulates are often on a higher level than the concepts of the knowledge base. For example, a *rule* created by the user is frequently translated into many rules for the inference engine.

Dark Matter Studio is built on top of Ontoprise's ontology engineering environment OntoStudio [156], which in turn is built on top of the Eclipse framework [7]. The Eclipse framework provides a plug-in framework that allows to seamlessly extend OntoStudio. The main reasoning component in Dark Matter Studio is Ontoprise's inference engine Ontobroker [50, 3] which utilizes Mathematica [138] for equation solving. The main hub for the knowledge formulation work is the annotation component, which is a further development of the annotation tool Ont-O-Mat [78] which includes support for semi-automatic annotation based on the KANTOO environment [121]. Graphical editors for the ontology were extended from the OntoStudio system; rules are created with a new graphical rule editor. Dark Matter Studio also includes dedicated editors for the formulation of process knowledge, explanations and tests. The verification support for Dark Matter Studio was conceived and largely implemented as part of this thesis. It consists of testing support (see section 4), debugging (see section 5.2.2) and anomaly detection heuristics (see section 6). The also includes a longer walkthrough showcasing some of the main components of Project Halo in section 5.2.3.

Evaluation Setup

The evaluation consisted of two parts, a knowledge formulation and a question answering part. In the first part six domain experts (DEs), two each for the domains of physics, chemistry and biology (senior students in these fields) were handed the system, trained on it for two weeks and were then asked to formalize some pages from a textbook in their field. In the second part of the evaluation domain experts were given DMS with one of the knowledge bases created during knowledge formulation. They were then given a number of questions for this domain that they should formalize as queries to get answers using DMS. Altogether the domain experts spent on average ca. 100 hours interacting with the system [81].



of **atoms**, the almost infinitesimally small building blocks of matter. Each element is composed of a unique kind of atom. We will see that the properties of matter relate not only to the kinds of atoms it contains (*composition*), but also to the arrangements of these atoms (*structure*).

Atoms can combine to form **molecules** in which two or more atoms are joined together in specific shapes. Throughout this text you will see molecules represented using colored spheres to show how their component atoms connect to each other (Figure 1.1). The color merely provides a convenient way to distinguish between the atoms of different elements. Molecules of ethanol and ethylene glycol, which are depicted in Figure 1.1, differ somewhat in composition. Ethanol contains one red sphere, which represents an oxygen atom, whereas ethylene glycol contains two.

Figure 3.7: *A part of the annotation perspective of Dark Matter Studio.*

	Time spent in DMS (hh:mm)
DE 1 (Physics)	106:56
DE 2 (Biology)	81:13
DE 3 (Biology)	128:25
DE 4 (Chemistry)	96:24
DE 5 (Chemistry)	84:32
DE 6 (Physics)	102:54

Table 3.5: *The time spend working with Dark Matter Studio for each of the domain experts.*

Based on [81, 79, 80] the following sections summarize the results from these two stages of the evaluation. Further evaluation results with respect to testing are presented in a later chapter (section 4.4).

Ontology Creation and Size

All domain experts were able to successfully create a taxonomy, relations and attributes. The size of the taxonomy varied between the domains, with the Physics DEs creating the fewest concepts and instances. The following table summarizes the size of the created ontologies. The table excludes some domain specific knowledge that was already in the systems when they were handed to the DEs. This *pump primed knowledge* was created by knowledge engineers and consists of some fundamentals for each area, such as the seven base SI-units for physics or the periodic table for chemistry.

	Concepts	Instances	Attributes	Relations
DE 1 (Physics)	14	6	34	11
DE 2 (Biology)	111	50	8	46
DE 3 (Biology)	172	278	8	24
DE 4 (Chemistry)	37	50	16	16
DE 5 (Chemistry)	75	93	55	45
DE 6 (Physics)	16	8	40	20

Table 3.6: *Measures of the size of the ontologies created by the domain experts.*

In the created ontologies the average number of sub-concepts was 3.1 per concept, ranging from 2.1 to 4.2 depending on the domain expert. The maximum depth of the taxonomy was between 3 and 11, with both biology domain experts creating the taxonomy with the largest depth (10 and 11).

	# Rules	# Disabled Rules
DE 1 (Physics)	50	3
DE 2 (Biology)	69	9
DE 3 (Biology)	130	7
DE 4 (Chemistry)	146	23
DE 5 (Chemistry)	175	175
DE 6 (Physics)	40	0

Table 3.8: *Number of the rules created and the rules deactivated at the end of the knowledge formulation phase.*

The created ontologies were mostly relatively lightweight, with domain experts making relatively little use of more advanced features such as sub-property relations or default values for properties.

	# sub-property	# mult. inheritance	# instance with mult. concepts	# default values
DE 1 (Physics)	0	1	0	0
DE 2 (Biology)	0	0	0	0
DE 3 (Biology)	9	0	0	1
DE 4 (Chemistry)	9	1	0	4
DE 5 (Chemistry)	3	0	0	0
DE 6 (Physics)	0	1	0	2

Table 3.7: *Usage of advanced features of ontologies.*

Rules and Processes

All domain experts were able to successfully use the graphical rule editor of DMS to create a large number of rules. The integration of the rule editor with the rest of DMS worked well and some domain experts where enthused to see their rules 'come to life' in the tests.

In order to prevent faulty rules from corrupting a large part of the knowledge base, the system contained an option to disable rules. This tool was used by all domain experts, in particular by domain expert 5 who disabled all her rules. DE 5 suffered from the problem that, while working well independently, her rules interacted in ways that broke the rule base in ways she was not able to diagnose. To have at least some success, DE 5 therefore disabled most of her rules, having only a few active rules at any given time; at the end of the evaluation she disabled all rules.

DMS also included editors for the graphical editing of knowledge about processes. However, because of initial problems with the process editing component during the evaluation, the domain experts did not have confidence in this component and used it only very sparingly or not at all.

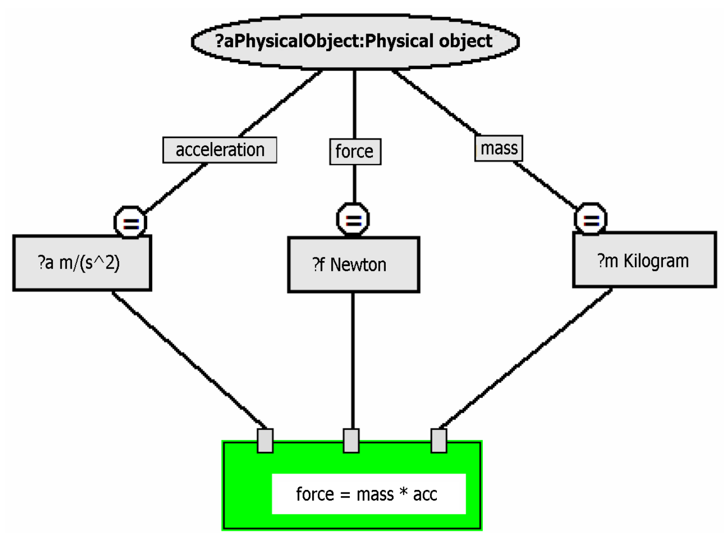


Figure 3.8: *Project Halo example rule from [81].*

Figure 3.8 shows an example rule [81] from a physics knowledge-base from Project Halo. This rule realizes the Newtons second law as a mathematical formula. The rule defines the context in which this equation applies and the parameters that are fed into the equation. This example rule can be paraphrased as: if ?aPhysicalObject is a Physical object and it has an acceleration of ?a m/s^2 and a force of ?f Newton and a mass of ?m Kilogram, the formula force = mass*acc can be applied. The relation between the attributes ?f, ?m and ?a and the variables in the equation (force, mass and acc) is defined by the position the lines from the attributes connect to the green box with the formula. The formula is shown in green indicating that it represents the conclusion from the rule, i.e. this is the rule head. This graphical representation of the rule is translated into multiple F-logic rules, ensuring that the equation can be used 'in any direction', i.e. that it can be used to compute accelaration from mass and force just as well as to compute mass or accelaration in cases where at the values of two other variables are known.

Testing

The testing component proved to be very popular with the domain experts. The graphical interface of the testing component was usable and all domain experts succeeded in creating a considerable number of tests. The main two conclusions from the evaluation with respect to testing were that [79]: 1) Only successfully tested rules have sufficient qualities to be used for question answering and 2) that it proved important to re-run all tests after only a small number of changes to ensure that these did not break the knowledge base in surprising places. The main problem for testing was the long time that running some of the tests took. This is not a problem of the debugging component but rather was caused by general performance problems of the inference engine [80]; nevertheless this caused users to use the debugging component less than would be ideal. More evaluation data about the testing component can be found in section 4.4.

Knowledge Base Quality and Question Answering

In the question formulation part of the evaluation, the quality of DMS and of the created knowledge bases was evaluated. To do this, the system together with a knowledge base was given to different domain experts that had to use it to formalize and answer a set of questions that hadn't been known before. In order for a question to be answered correctly, the knowledge base needs to be correct, needs to encompass the needed knowledge and the question formulation domain expert (QF DE) needed to correctly formalize the question; i.e. to formalize it in a way that it yielded an answer and also accurately reflected the question. An incorrect formalization of the question could again depend on an error by the QF DE, knowledge missing from the knowledge base[9] or a limited expressiveness of the question formulation tool.

The question answering performance for questions judged to be fully adequate is shown in the table below. The numbers show the count of results that were fully, nearly or not adequate; only the results for fully adequate queries are shown. It can be seen that only the Physics DEs succeeded to formulate and correctly answer a considerable number of questions.

[9]e.g. the knowledge base lacked the terminology to represent the relations that the question asked about

	# Fully Adequate	# Nearly Adequate	# Inadequate
DE 1 (Physics)	4	0	3
DE 2 (Biology)	1	0	0
DE 3 (Biology)	0	0	0
DE 4 (Chemistry)	0	0	0
DE 5 (Chemistry)	0	0	4
DE 6 (Physics)	7	0	2

Table 3.9: *Rating of answers for fully adequately formulated queries.*

A larger number of questions could be formalized nearly adequately. The question answering performance on these queries is shown below. Again the table shows the number of results that were fully, nearly or not adequate for nearly adequate query formulation.

	# Fully Adequate	# Nearly Adequate	# Inadequate
DE 1 (Physics)	6	0	0
DE 2 (Biology)	0	1	0
DE 3 (Biology)	1	1	1
DE 4 (Chemistry)	0	0	5
DE 5 (Chemistry)	1	0	4
DE 6 (Physics)	7	0	6

Table 3.10: *Rating of answers for nearly adequately formulated queries.*

Knowledge missing from the knowledge base was the most common reason for questions that could not be formalized adequately; this affected 62% of the cases. Another important problem affecting 27% questions was the limited expressiveness of the question formulation tool. The remaining cases were caused by limits in reasoning and errors in question formulation [79].

Evaluation Summary

Overall the very challenging evaluation setup that required domain experts to model knowledge that could successfully answer novel questions posed by other domain experts showed some success, in particular in the physics domain. At the same time, however, most questions could not be correctly answered, showing major limitations of DMS. The final implementation report [81] blamed this mostly on three clusters of problems:

- **Question Formulation Workflow:** An underestimation of the difficulty of formalizing question.

- **DMS not restrictive and not transparent enough:** The software allowed domain experts to build models that would break the reasoner in ways not transparent to the user.
- **Project Management and Project Team:** Problems in the project structure that led to late architectural changes and late system integration.

Proposals were made to address these problems [81], however, these were not realized due to Project Halo's focus shifting towards web-based systems.

For this thesis Project Halo provided input for the analysis and design. Further, some implementations were also integrated into Dark Matter Studio and evaluated in the context of this project.

3.1.3 Smaller Projects

Two smaller rule base development projects form a further input for the analysis. Online Forecast - an application of rule bases for human resource reporting - is presented first. The second project is F-Verify, an anomaly detection rule base.

Online Forecast

Online Forecast [131] 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 at Daimler AG was re-created as rule based system. This project was performed by a junior developer with little prior experience with rule based systems who spent approximately 5 person months creating it.

The application processed human resource data stored in a relational database. One set of mapping rules was responsible for mapping the data from the database to a human resource ontology. A second set of rules aggregated data into higher-level concepts and supported users in generating reports such as 'how many people left the company last month because they have to serve compulsory military service'. A number of integrity rules were included to support users in maintaining data integrity, and explanation facilities were realized prototypically for a small part of the domain. Altogether the knowledge base consisted of 21 concepts with 35 relations and 100 rules with up to 38 body atoms. The instance data used during the creation of the system consisted of anonymized data from roughly 15000 employees, in the end represented by almost one million facts.

Online Forecast used F-logic as representation language and the Ontobroker inference engine for reasoning. The commercial ontology engineering environment OntoStudio [156] was used as editor. The project was realized with a lightweight, iterative process. It began with the creation of the domain ontology in close cooperation with the domain experts. In the next steps informal descriptions of rules were created based on the legacy application, these were validated with the domain experts and then implemented. The results from applying the created rules to the database were compared to the results from the legacy application and all diverging results were discussed with domain experts - this step uncovered multiple errors in the legacy application, as well as some in the rule base. The rules were adapted based on this feedback and the result was again presented to the domain experts.

As a whole the project was a success in so far as it could show that rules can be used to re-implement the reporting application with higher precision and additional functionality in the form of explanations and integrity constraints that should enhance understandability and maintenance. The integrity constraints uncovered errors in the existing data, thereby showing their usefulness. However, the application was not used productively, because it did not easily fit within the existing IT-infrastructure and because the skill sets needed for maintaining it were not available at the company.

For this thesis this project forms an importance input for analysis and design. During Online Forecast the problem of insufficient debugging support became very clear. In particular the debugging of the rule interactions turned out to be difficult because many test-queries simply returned no result, giving very little information that could aid error localization.

F-Verify

F-Verify (see also section 6) was a project to create a rule base to support verification activities. 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, i.e. that reason about the rules themselves. This project was done by a developer with experience in creating rule bases and took 3 months to complete.

F-Verify was created in F-logic and also used for the diagnosis of F-logic rule bases. It used an informal iterative process: new heuristics would be developed and defined in natural language, then implemented and added to the rule base. This was repeated multiple times. The domain model (the reified rules) existed already and was reused. Altogether the rule base consisted of 60 rules (examples are included in section 6). It was built in the style of constraints and included facilities to explain the problems in the rule base to end users.

This project was created to be used in later version of DMS. The rule base was successfully created, but was never deployed because of the reasoning performance problems affecting DMS [81].

The experience from this project further established to problem of tools support for fault detection and prevention in rule based systems. In particular the lack of debugging support and a testing infrastructure hampered development. Another observation from this project was, that with rule based systems the error localization is made very difficult by some errors caused by interactions between rules in parts of the rule bases that are completely unrelated from a user's point of view (further explained below as the Problem of Interconnection).

3.2 Analysis

This section is dedicated to the analysis of debugging challenge based on the empirical basis introduced in the previous section. It starts with the explanation of the debugging challenge and presentation of evidence for its existence. Next, the fault lifecycle is introduced as the model developed to guide and summarize the analysis. After that five problem hypotheses are presented for the explanation of the debugging challenge. Finally a section introduces major trends in rule base development that must also be considered in any approaches that attempt to tackle the debugging challenge. All results from the analysis section are then aggregated and used to derive concrete approaches in the following design section.

3.2.1 The Debugging Challenge

The assumption of rule based systems as simpler than 'conventional' programs 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.

To see whether this simplicity assumption holds in practice, the developer survey (section 3.1.1) included a question that asked participants to compare rule base development to the development of procedural and object oriented programs. The results show (figure 3.9) that these developers indeed see the advantages in ease of change, reuse and understandability that were expected from rule based (and declarative) knowledge and program specification.

However, at the same time a majority of rule base developers sees rule base development as inferior with respect to debugging; participants of the survey see debugging of rule based systems as actually more difficult than the debugging of conventional software. And, in a different question, rank it as the issue most hindering the development of rule based systems (see section 3.1.1). This observation was also corroborated in the three rule base development projects described earlier. In these projects also the overall question preventing, finding and removing faults proved to be insufficiently supported.

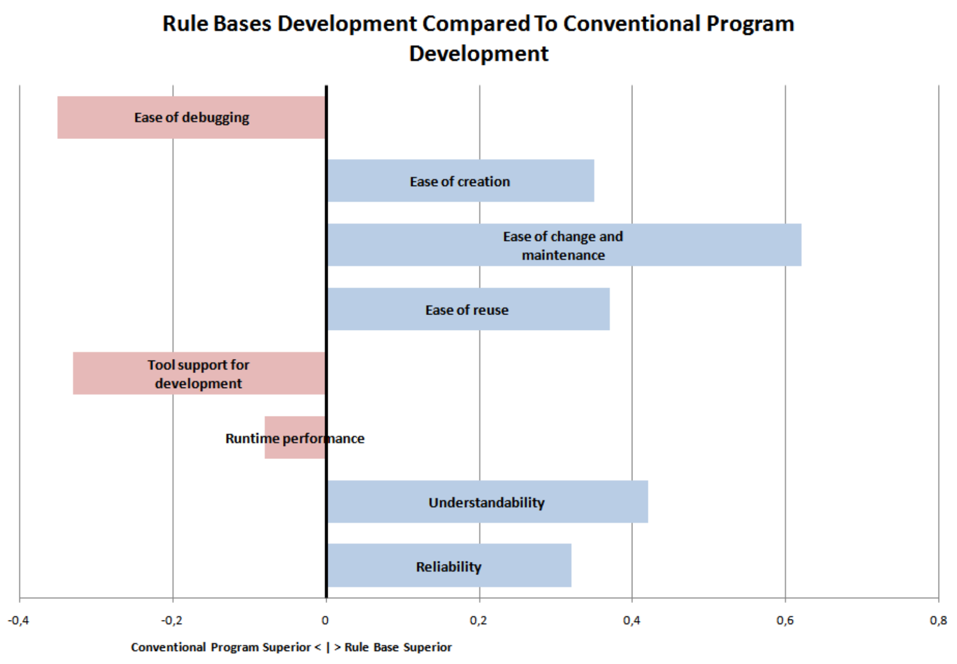


Figure 3.9: *The aggregated results from the survey questions asking participants to compare rule base and 'conventional' software development*

3.2.2 Fault Lifecycle

In order to organize and guide analysis and design, the fault lifecycle was developed. It an is overall conceptual model that shows the different processes that affect implementation faults in any software system's development. The fault lifecycle will be used in the consecutive sections of this chapter.

During the development of a software many **actions** are made. Some of these will be **mistakes** - inappropriate actions during software development that may later cause a program to fail. A mistake may result in a **fault**, an incorrect step, process or data definition in a computer program [84]. Finally a fault may manifest itself in a computation that involves - or should have involved - instructions affected by the software fault and that results in an **error**, a difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition [84]. None of these transitions is inevitable; a mistake may be caught before it results in a faulty implementation and a fault may be neutralized by later code or may never be exercised in the running program. At the same time many techniques are routinely applied to reduce the number of faults in a program. Static Analysis techniques such as code review, formal verification or anomaly detection identify faults in the program code. Testing techniques execute the program in a controlled environment, discovering some of the errors that otherwise may appear in a production environment. Finally, debugging techniques are used to identify a fault based on an error detected in testing.

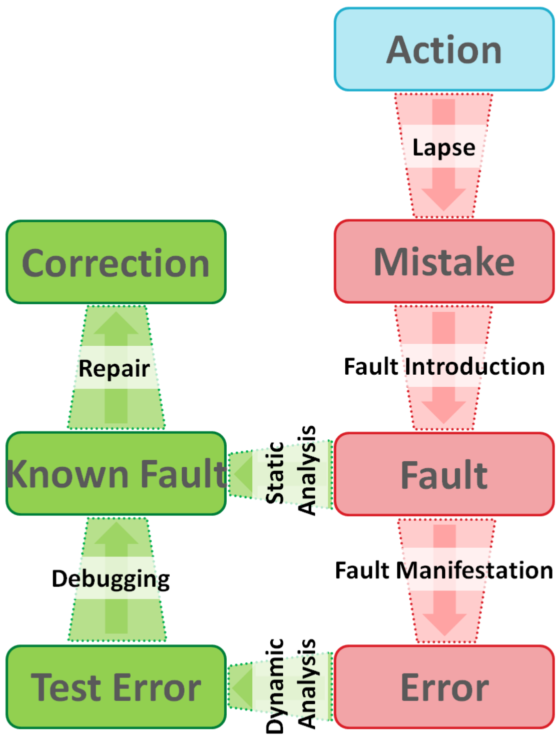


Figure 3.10: *The Fault Lifecycle and its transitions*

It is important to note, that the notions of action, mistake, fault and error are applicable at many levels of the software development process, such as software design, software architecture or requirements engineering. In this thesis, however, the fault lifecycle is used exclusively with respect to the actual implementation.

Every transition in the fault lifecycle model is a potential trouble spot that can lead to a program failing more often; at the same time each transitions also offers a lever to make the creation of software with less faults more effective. A short explanation and two examples for each of the transitions shall make this and the Fault Lifecycle clearer:

- **Lapse:** This transition reflects lapses in the action of developers that mean a mistake is made. The strength of this transition (i.e. the proportion of actions that are mistakes) is affected for example by the experience of the developers creating a rule base. Inexperienced developers will make more mistakes and ultimately create rule bases with more faults. On the other hand, visualizations or good architectural metaphors improve a developers understanding of the program and make it easier to make the right action.

- **Fault Introduction:** A mistake by a developer may introduce an incorrect step, process or data definition into the computer program. However, appropriate processes or tools may also catch the mistake before its introduced into the code base. Using only a simple text editor instead of one with syntax highlighting increases the chances that mistakes lead to faults in the code. Pair programming, however, is one technique thought to positively influence this transition; here a second developer immediately questions all actions and can hence spot mistakes early.
- **Fault Manifestation:** A fault may manifest itself in a computation that did involve, or should have involved a statement affected by the fault; this results in an error. Not all faults cause errors and one fault may manifest itself in many different program behaviors. In rule based systems the number of errors caused by a fault is influenced by the structure of the inference engine as well as the semantics used. For example using a rule base with a semantics that is only defined for globally stratified models means that a single fault can - by making the rule base unstratisfiable - cause a large number of errors. On the other hand an inference engine that is able to recover from some errors by automatically removing a rule from consideration will produce less errors for a given number of faults.
- **Static Analysis:** These techniques examine the program without running it, they try to identify faults through a careful analysis of the program code. Detection of anomalies and bug patterns, formal proofs of properties of a program and code review are techniques used for this transition. Most of these techniques cannot directly identify a fault [135], but rather they find anomalies that are then used by the developers as hints for fault identification. The absence of a formal specification is one factor aggravating the static detection of faults; on the other hand the availability of a type system or an ontology describing the terms in the rules improves it.
- **Dynamic Analysis:** Dynamic analysis techniques execute the program to identify anomalies in the behavior of the program. The most common used method for this transition is testing, however, run time monitoring to uncover interim faulty states is also important. A large difference between the environment used for testing and the runtime environment of a system is one trouble spot that will affect the efficiency of this transition. An infrastructure for automatic regression

tests may improve it.

- **Debugging:** Through debugging an error observed under controlled circumstances is investigated to identify the fault that caused it. Debugging is influenced positively for example by good error messages and negatively by the absence of debugging tools.
- **Repair:** Once a fault has been identified the developer must find a way to correct without impairing the rest of the program. A hard to understand program or unsuitable editors make correcting faults difficult, automatically generated change proposals could make it easier.

It is important to note that the Fault Lifecycle developed here represents a simplification of the fault releated activities in software development; that in modern technologies are sometimes transcending its borders. For example by examining all possible execution paths of a program, model checking transcends the difference between static and dynamic analysis. Also, many modern development environments already include anomaly detection and even regression testing before anything is added to the common codebase; in this sense these systems harness static and dynamic analysis already to improve the 'fault introduction' transition.

3.2.3 Problem Hypotheses

This section introduces six problem hypotheses for the explanation of the debugging challenge; for explaining the apparent mismatch between the simplicity of rule bases and the observed difficulty of debugging. The problem hypotheses are: (1) One Rule Fallacy and the Problem of Terminology, (2) the Problem of Opacity, (3) The Problem of Interconnection, 4) The No-Result-Case and The Problem of Error Reporting, (5) the Problem of Procedural Debugging and finally (6) The Problem of Tool Support. Each of these problems is described in a short section below.

The One Rule Fallacy and the Problem of Terminology

Declarative statements such as rules are often assumed to be easier reusable and more modular, because they are true in broader contexts than program statements can be used. As McCarthy put it in 1959 [107]:

> *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 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.

Based on these sentiments the one rule fallacy goes as follows:

> One rule is simple to create and
> the combination of declarative rules is handled automatically
> by the inference engine
>
> **hence**, entire rule bases are simply 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. In particular for rules in a rule set to work together, all rules need to be consistent in the domain formalization and the use of terminology. For example it must not happen:

- That one rule uses a relation $part$ while another one uses $part\_of$.
- That one rule understands a $parent$ relation biologically, while another understands it socially.
- That one rule processes a number as inch, while another processes it as millimeter.

Rule based systems hold the promise to allow the automatic recombination of rules to tackle even problems not directly envisioned during the creation of the rule base. However, this depends on ensuring the consistent formalization and use of terminology across the rule base.

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. At the same time this points to the Problem of Terminology as one explanation for the difficulty in debugging rule bases. The problem of terminology makes the rule base harder to understand, makes it harder to identify, avoid and correct mistakes based on only a local view of the rule base. In the fault lifecycle the problem of terminology hence affects mostly the *lapse* transition; also other transitions (such as debugging and repair) are also affected.

The Problem of Opacity

In the authors experience failed interactions between rules are the most important source of errors during rule base creation. At the same time, however, it is this interaction between rules that is commonly not shown; that is opaque to the user. The only common way to explicitly show these interactions between the rules is in a prooftree (see section 4.2.1) of a successful query to the rule base. This stands in contrast to imperative programming, where 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.

Not having an overall view on the rule base mainly affects the ability of developers to correctly edit the rule base, because it becomes hard, for example to see the consequences of changes or to gain confidence in the completeness of a rule base. The Problem of Opacity also affects the developer's ability to identify faults in rule bases.

The Problem of Interconnection

Because rule interactions are managed by the inference engine, everything is potentially relevant to everything else[10]. This complicates error localization for the user, because errors appear in seemingly unrelated parts of the rule base.

As a very simple example consider the following rule base:

$$hot(X) \leftarrow part\_of(Chile, X)$$
# fault - should be part_of(Chili,X)

..

$$part\_of(Chile, SouthAmerica)$$

In this example a rule that deduces that something is hot based on it having chili as part, is changed by a typo to deduce that something is hot if the country Chile is part of it. This simple error then introduces a connection to a part of the rule base that deals with countries and that - from the user's point of view - is absolutely unrelated.

More typical than this example, however, are cases that do not lead to wrong conclusions but rather merely lead the inference engine to try one path that ends in a rule cycle or in some kind of error. One example would be that a rule causes a whole rule base to become unstratisfiable (see section 2.1.5), another that it leads to a rule being tried in unexpected circumstances that then lead to errors due to the wrong usage of built-ins.

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

The Problem of Interconnection appeared frequently during the three rule base development projects and is visible for example from these statements by two Project Halo domain experts [69][11]: *"... there are still rules that do not work together, even though there is no connection between them."* and *"... some rules cause problems in an inexplicable way"*. These statements were uttered in response to errors caused by rule interactions, but interactions the users were not aware of or (as is evident from the first statement) even (falsely) confident that didn't exist. The main transition affected by the Problem of Interconnection is the debugging connection that represents the identification of faults based on observed errors. This identification is made more difficult by the Problem of Interconnection. Another consequence of the Problem of Interconnection is that even a single fault can cause many errors across a large part of a rule base's functionality.

The No-Result Case And the Problem of Error Reporting

In the three rule base development projects by far the most common error, by far the most common symptom of a fault, was that a query (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. A problem made more significant by the fact that some debugging tools for rule based systems only work in cases where there is a result[12]. 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 survey of rule base developers also included a question asking which kinds of errors were encountered, never, sometimes or frequently. Overall it showed that 64% of developers frequently and a further 34% sometimes encounter a fault showing itself in a missing conclusion. Looking only at those development projects that used one of the environments classified as *declarative*, i.e. that used rule languages most closely resembling the kind of rule languages under discussion in this thesis, the no-result case even becomes the error most frequently encountered (see figure 3.11).

An experiment with randomly seeded faults further confirmed this observation. In this experiment two small rule bases were randomly altered[13] and the result for one query each was compared to the expected value. The

[11]Translated from German by the author.

[12]In particular this affects all traditional explanation approaches (see section 5.4.2).

[13]The details of the seedings process are described in section 5.3.

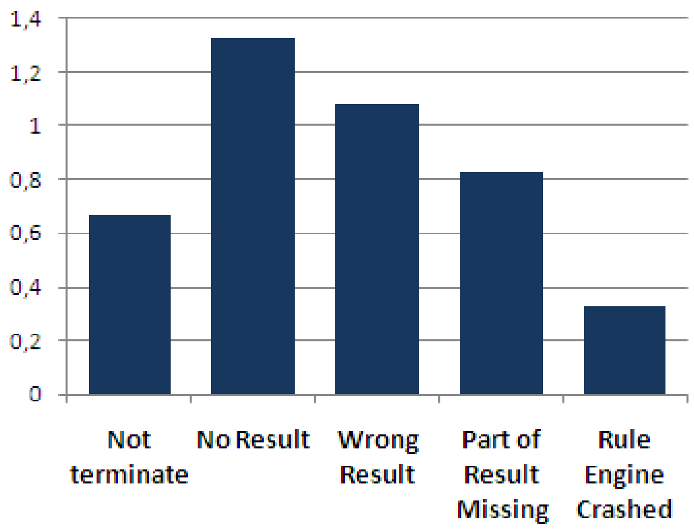


Figure 3.11: *Symptoms of faults in declarative rule bases, based on the survey of developers*

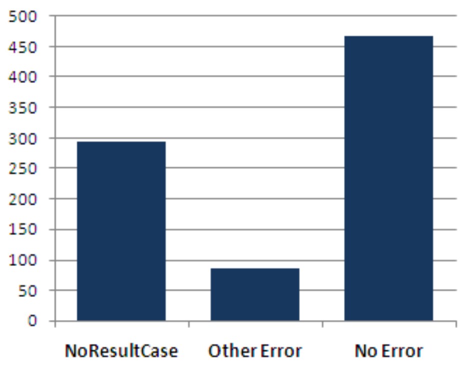


Figure 3.12: *Sympoms from randomly seeded faults*

largest proportion of seeded faults did not result in any error (see Figure 3.12), but of the errors that where caused, 75% where no-result cases.

In the fault lifecycle the Problem of Error Reporting affects the identification of faults based on errors - this is hampered by cases where the error gives no clue for the identification of the fault.

The Problem of Procedural Debugging

The overwhelming majority of tools used for the debugging of rule based system are based on the procedural or imperative debugging paradigm (see figure 3.13 and the discussion in 3.1.1). 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 (a longer description of this debugging paradigm is given in 5.4.1).

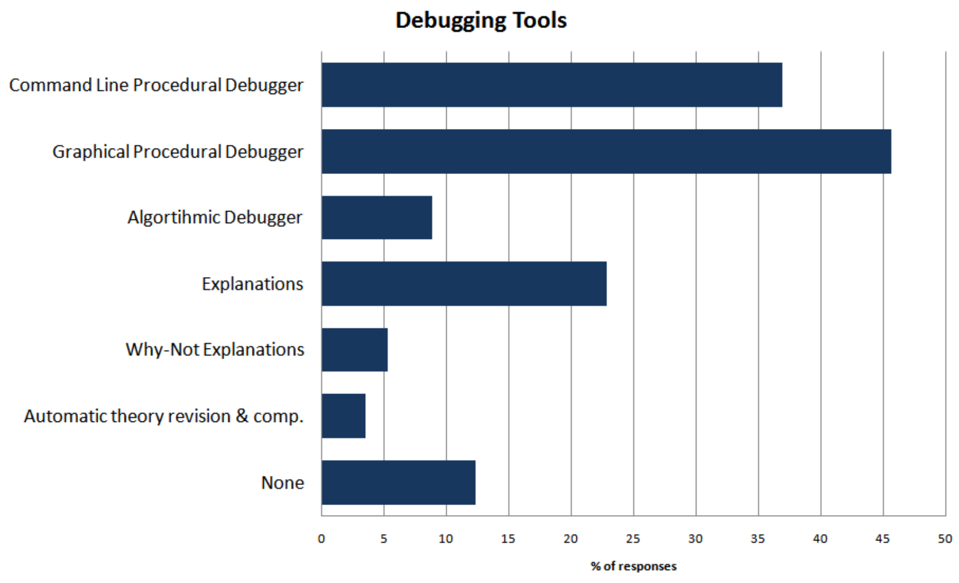


Figure 3.13: *Tools used for debugging*

However, as a declarative representation 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. In this way the debugging of rule bases breaks the declarative paradigm - they 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.

In the fault lifecycle the Problem of Procedural Debugging affects the transition from test error to fault: the debugging transition.

The Problem of Tool Support

Finally, 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 of the fact that in recent years the percentage of applications built with imperative programming languages was much larger than those built with rule languages. This discrepancy is also visible in the results from the survey, where (in addition to debugging support) the lacking refinement of development tools emerged as an area where rule based system development lags behind 'conventional' software development (see figure 3.9).

Less refined tool support is something that affects all transitions in the fault lifecycle. Within the context of the three projects observed, the dynamic analysis transition was most affected. Prior to the work conducted for this thesis, testing - the most commonly used verification method for rule bases (see figure 3.4) - was not supported at all for F-logic/LP-O rule bases, e.g. there was no infrastructure to store and run test cases, no user interface to create tests and no test coverage measure.

3.2.4 Changing Environments

In this section, four major trends in rule base development are presented as another input for the design. To tackle the debugging challenge, tools need to work within the scope and constraints of modern rule based development processes characterized by these trends; the six problems described above must be tacked in this environment.

Based on literature research and on the survey of rule base developers, this section looks at these changing processes and at the requirements for debuggers that follow from them.

End User Programmers and Domain Experts

The amount of software in society increases continually, and more and more people are involved in its creation. The creation of software artifacts used to be the very specialized profession of a few thousand experts worldwide, but has now become a set of skills possessed to some degree by tens of millions of people. Within this group an increasing role is played by end user programmers - people that are usually trained for a non-programming area and just need a program, script or spreadsheet as a tool for some task.

For the US it is estimated that there are at least four times as many end user programmers as professional programmers [148]; with estimates for the number of end user programmers ranging from 11 million [148] up to 55 million [18]. End user programmers are particularly important for web related development, because web developers are known to have an even higher percentage of end user programmers [139, 82].

This rise in the number of end user programmers means, that the average developer is not trained as well as the knowledge engineers that created the first expert systems. End user programmers usually can't justify making an investment in programming training comparable to that of professional programmers.

The survey of developers also found that the average rule base development projects includes 1.5 domain experts that create rules themselves and 1.7 domain experts that were involved in verification and validation. Slightly more than half of the projects included at least one domain expert that created rules.

Lesser training and experience of the developers is a challenge for all parts of the Fault Lifecycle [116], most importantly, however, it means that simply more mistakes get made.

Embedded Rule Systems and Embedded Procedures

Current rule based systems are commonly not stand alone systems but are rather embedded in a context of other systems, probably most of them built in a different language. This integration goes both ways: on the one hand rule bases are including and calling built-ins that are often implemented in a procedural language. On the other hand rule bases are part of a larger, conventional program. In particular in the business rule community it is common [110] that rules are only used to represent the core, mutable business process aspects of a system, while the rest of the application is still realized with an object oriented programming language.

This property is also visible in the developer survey described above. It found that on average the rule bases took 9 person months to develop while the overall application needed 60 person month; almost 6 times the effort.

This embedded nature of rule based systems means that on the one hand the verification support must be aware and support problems that occur on the boundaries of the rule base; that involve calls to other programs possible created in a different paradigm. This may complicate fault localization based on test results as well as fault identification based on static analysis. Secondly, however, it also means that rule bases tend to be relatively small when compared to programs created in 'conventional' programming paradigms; simplifying the creation and editing somewhat.

Agile, Iterative and Lightweight Methods

Recent years also saw the increasing adoption and acceptance of iterative and evolutionary development methodologies [97] - they are now widely believed to be superior to waterfall-like models [102].

A high prevalence of agile and iterative methods was also found in the developer survey. Agile and iterative methods were used in more than 50% of the projects with more than 10 person months of effort. A further 23% of the projects worked without following any specific methodology.

The first corollary of this observation is that the verification support for these systems cannot rely on the availability of formal specifications of any kind; an impediment for many static analysis methods. A second likely consequence is a higher prevalence of implementation mistakes - since with many of these methods, architecture and design decisions are taken on the fly during the implementation of the software. In addition, these design and architecture decisions are often repeated and changed, because agile methodologies encourage developers to make these decisions based on the scope of the current iteration, not on the expected scope of the final application. Doing these design and architecture decisions 'on the fly' reduces the risk of costly requirements, design and (up-front) architecture mistakes, however, at the same time it means that many of these mistakes are then made during implementation.

Rule Interchange, the Semantic Web and Ontologies

Emerging rule interchange[14] [19] and the Semantic Web [91] are also posing new challenges for verification support, but are also offering new opportunities. Rule interchange and the Semantic Web pose challenges, because they may lead to larger rule bases created without strong central support. Further rule interchange means that people will often be faced with verifying a software system including at least a part that was created externally, making things like the Problem of Interconnection even more daunting.

The Semantic Web, however, is also opening up new opportunities for better debugging tools. The purely declarative nature[15] of many rule languages under consideration for the Semantic Web (e.g. WSML, SWRL and F-logic) make it seem possible, that creation and debugging of rule bases can be done without needing to learn about the evaluation strategy of the inference engine. Also, the availability of explicit ontologies for the terms used in the rules can offer additional information to better support the debugging process.

Within the fault lifecycle these new languages are hence offering new ways for the static identification of faults through the use of ontologies; e.g. identifying rules that cannot fire can be aided by knowing that two concepts used are disjunctive, do not share any instances. At the same time rule interchange and the reuse of rule bases on the Semantic Web are also posing new challenges for fault identification, for the fault identification from test errors as well as for the static identification of faults.

[14]I.e. the trend to create rule interchange formats in order to establish rule sets as tradeable artefacts that can be exchanged between companies.

[15]In contrast to, e.g., Prolog or the rule languages used in most business rule management systems.

3.3 Design

This section aggregates the results from the analysis and gives a very coarse design for tool support that can tackle the debugging challenge of rule based systems. The design sketched in this section is then further elaborated, implemented and evaluated in the remaining chapters of this thesis.

In section 3.3.1 this chapter starts with an aggregation of the problems and challenges identified in the analysis. Then, based on literature research and on experiences from the rule base development projects described at the beginning of this chapter, four overall design principles are presented in section 3.3.2. Finally, adressing the problems identified in the analysis in the previous section, four concrete in the areas of testing, debugging, static analysis, and visualization are shortly sketched (section 3.3.3). Their elaboration is the topic of successive chapters.

3.3.1 Choke Points and Requirements

Aggregating the results from the analysis using the fault lifecycle one can identify the transitions most affected by the properties of rule based systems (see figure 3.14).

This aggregation shows the debugging transition as affected by the largest number of problems. In detail the affected transitions are

- The **debugging** transition from test error to known fault is affected by five of the ten problems. The Problem of Interconnection (3) hampers fault localization because it leads to errors appearing in surprising places. The Problem of Error Localization or the no-result-case (4) makes it hard to diagnose faults, because in many cases the error is simply an empty result case, giving the user no information on where to start the debugging process. The Problem of Procedural Debugging (5) means that developers have to learn about the *how* of the computation, about the internal structure of the inference engine to use state of the art debuggers. The embedded nature of rule bases (8) leads to hard to diagnose faults at the boundaries of different programming paradigms, but also means that rule bases tend to be relatively small. Finally the Semantic Web and rule interchange (10) mean that debugging may include rule bases developed independently, without central control, meaning that often the developer doing the debugging has to learn about the rule base at the same time.
- The **lapse** transition (proportion of actions that are mistakes) is mainly affected by the Problem of Terminology, Opacity and by the

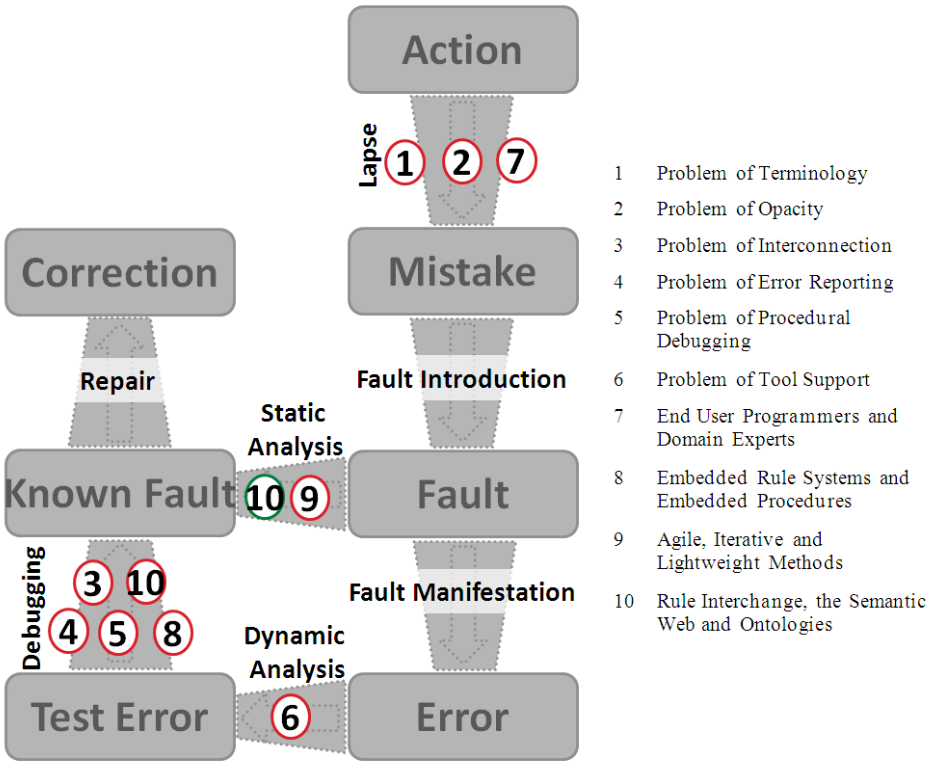

Action
Lapse
1
2
7
Mistake
Fault Introduction
Fault
Fault Manifestation
Error
Correction
Repair
Known Fault
Static Analysis
10
9
Debugging
3
10
4
5
8
Test Error
Dynamic Analysis
6
1 Problem of Terminology
2 Problem of Opacity
3 Problem of Interconnection
4 Problem of Error Reporting
5 Problem of Procedural Debugging
6 Problem of Tool Support
7 End User Programmers and Domain Experts
8 Embedded Rule Systems and Embedded Procedures
9 Agile, Iterative and Lightweight Methods
10 Rule Interchange, the Semantic Web and Ontologies

Figure 3.14: *The fault lifecylce with an overview of the analysis. The red circles indicate the challenges for each transition, the one green circle indicates the new opportunities afforded for static analysis through the use of ontologies (see section 3.2.4).*

rise of end user programmers. The first two are concerned with the connection between rules that are not easily visible. The Problem of Opacity (2) with the fact that the overall program is determined by rule interactions normally not shown to the programmer. The Problem of Terminology (1) with the observation that, while often seen as self-contained entities - rules are in fact dependent on a domain formalization consistent across all rules. Finally, the rise of end user programming (7) means that less trained people are creating the rule bases.

- The **static analysis** transition is hampered by the increasing use of agile processes (9) that mean that there are little or none formal specifications. At the same time, however, the use of ontologies to describe the terms used in rules may also offer new ways to improve this transition.
- Finally the **dynamic analysis** transition was the one - in the context of the projects examined for this thesis - most affected by the problem of missing or immature tool support.

To address these problem areas the following section presents four overall design principles that, when embodied by tools, can tackle some of these points. The last section in this chapter then presents four concrete approaches that directly tackle the choke points presented above.

3.3.2 Design Principles

The author has identified four core principles to guide the building of better tool support for the creation of rule bases. These principles were 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 were the most successful tools employed. In cases where quick feedback during knowledge formulation could not be given[16], this was reflected immediately in erroneous

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

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 [145, 146]. 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.

- **Visibility:** To show the hidden structure of (potential) rule interactions at every opportunity. Visibility is included as a direct counteragent for the problem of opacity and the Problem of 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. Declarativity of all development tools is a prerequisite to realize the potential of reduced complexity offered by declarative programming languages. This principle is motivated by the Problem of Procedural Debugging.
- **Modularization:** [86, 4, 74, 109] 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. By imposing some user defined structure on the rule base, modularization becomes one way to tackle the problems caused by the invisible nature of the interactions in the rule base. However, modularization is also only a mixed blessing for rule based systems. Any introduction of modules that restrict the potential of rules to interact also means some decrease in flexibility of the rule base - that the rule base is less able to adapt to use cases not directly envisioned during the rule base's creation. Modules also introduce another layer of complexity - create a new class of faults, where expected conclusions are not reached because of faulty assignment of facts and rules to modules.

3.3.3 Concrete Approaches

Based on the transitions identified as particularly problematic as well as best practices described in literature (e.g. [135, 75, 162]) a number of concrete approaches is developed in this thesis (the numbers in brackets refer back to figure 3.14):

- For the **testing** transition a testing framework for F-logic was devised, implemented and evaluated within the context of Project Halo. In this context a novel test coverage notion for normal logic programs was developed. This tackles the Problem of Tool support (6) for the dynamic analysis transition that is of profound importance to almost every software development process. This is described in detail in section 4.

- For the **debugging** transition Explorative Debugging was developed, implemented and evaluated as a new approach to the declarative debugging for rule bases. Explorative debugging together with its innovative components of Mutex-Prooftrees and Mutex-Depends-on-Graph, is described in detail in section 5. The debugging approach centers particularly on making purely declarative debugging a reality (tackling the Problem of Procedural Debugging (5)) and on aiding the user in Error Localization in the no-result-case (4,3). The Explorative Debugging approach further facilitates learning while debugging - partly tackling the problem caused by rule interchange (10). Finally the implementations described in section 5 also include specific tools to enable the developers to better deal with imperative procedures embedded in the rule base (8).

- To support users with the challenges of the **lapse transition**, a novel visualization approach for rule bases was developed. This approach tackles the problems caused by the lack of developer defined structure by using the runtime rule interactions to create an overall visualization of the rule base. By making the hidden structure of the rule base visible, this approach directly tackled the Problem of Terminology (1) and the Problem of Opacity; it is also an important aid in particular for end user programmers (7). This approach is described in detail in section 7.

- Finally a set of anomaly detection heuristics was developed, to utilize the novel opportunities offered by terms described in ontologies (10) as well as to tackle the static analysis challenge caused by the absence of formal specifications and requirements in development projects using agile development methods (9). This is described in detail in section 6.

Chapter 4

Testing

"Testing is the process of executing a program with the intention of finding errors." [117], or more elaborate [84]:

> (1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.

For the purpose of this thesis, I extend even the relatively broad first definition, arriving at

> Testing is the process of executing a program with intention of uncovering errors or gaining a better understanding of the program.

The extension is needed to reflect the particularities of domain expert driven rule base development- we found that one of the most important uses of the testing component was to help developers better understand how rule bases work and how they can create them.

Testing is easily the most widely deployed measure to find and remove faults in software. For example, in a survey Perry [130] found that on average almost a quarter of a project's development budget was allocated to testing. The survey of rule base development processes described in section 3.1.1 also found that testing is by far the most common measure taken to remove faults from rule base.

Despite this importance, however, there are still clear deficiencies in the technical and conceptual support for tests of normal logic rule bases. For one, there is no generally accepted notion about the makeup of test cases and there are no established test coverage metrics. In the rule base development environment that was used as basis for Project Halo, there was even no testing support at all; when the work on this testing framework commenced in 2002, no testing framework could be found for either F-logic or normal logic programs.

The work underpinning this chapter aimed to address these questions in the context of Project Halo. The main parts of this work are:

- The definition of testing concepts for normal logic programs.
- The definition of a new test coverage measure based on the notion of least general generalization and prooftrees.
- The implementation of the testing concepts in a framework that enables domain experts to create tests.
- An evaluation of this test framework with domain experts. The evaluation showed that the testing concepts and user interface succeed in enabling domain experts to create and run a large number of tests. This enabled the domain experts to learn rule base development and to create working rule bases.

The chapter is structured along these lines, with one section for each of these points, followed by a discussion of prior work and a conclusion.

It is important to note that the implementation and evaluation of the work presented here was not done exclusively by the author. The author implemented the overall test framework and its user interface, however, the graphical test query and test fact editors as well as the stylized english view (all adaptations of tools also used for purposes other than testing), where designed and specified by the author, but implemented by different developers. Further, the evaluation of the testing framework within DarkMatterStudio was done by an independent third party; the author was not present. All conceptual work presented was done by the author.

4.1 Test Cases and Test Sets

This section introduces the main concepts of the testing framework proposed in this thesis; these are the notions of **test case**, **test facts** and **test sets**. To make these concepts accessible a simple example will be developed throughout this chapter. The rule base that is to be tested in this example consists of only two rules and is shown below. The rules allow for the conclusion that someone is a parent, if he is male and has a child.

$$parent(Y) \leftarrow fatherOf(Y, X) \text{ \# Rule parent 1}$$
$$fatherOf(A, X) \leftarrow male(A), child(A, X) \text{ \# Rule fatherOf}$$

In general a test set can be thought of as similar to a physics question in an exam. It includes some description of a situation and a number of questions that should be answered using the rule base with respect to the situation described. An example would be a situation describing a mass moving frictionless at a certain speed. There could then be a number of queries about this situation - such as 'what is the impulse of this mass' - that should be processed using the rule base.

Definition 1 (Test Facts) *A possibly empty set $T_f = \{f_1, ...f_n\}$ where $f_1, ..., f_n$ are facts and for all $f_x \in T_f$ it holds that $f_x \notin P$, where P is the normal logic program that is to be tested.*

Test facts describe the situation that the query will be run against; they describe the input that is to be processed by the rule base. For the rule base example described above, test facts T_{f1} could be:

$$male(peter)$$
$$child(peter, mike)$$

A test case then consists of the actual query and the expected answer. It is defined as follows:

Definition 2 (Test Case) *A tuple $T_C = (q, R, l)$, where q is a query consisting of a set of literals $q = \{L_1, ..., L_n\}$, R is a set of substitutions $R = \{\theta_1, ..., \theta_m\}$ and l is a label that is either $true$ or $false$.*

For the example a test case T_{c1} could be defined as follows: $q_1 = \{parent(Y)\}$, $R_1 = \{\{Y/peter\}\}$ and $l = true$. This test case contains the query asking for all entities known to be $parent$ and defines the correct answer as consisting of only the binding of Y to $peter$. The label is needed for cases where the query is ground - in these cases R is always an empty set and hence does not allow to differentiate between a query that follows from the program and one that does not[1].

[1]Consider for example the query $q_1 = \{parent(peter)\}$ - the set of subsitutions for this query is always empty, irrespective of whether $parent(peter)$ follows from the knowledge base or not. Posing a ground query to a rule base may seem like a futile exercise, since the expected answer must be known to create the query; however, to *test* a rule base, it is important to check whether an expected conclusion is indeed reached; entailed by the rule base.

A test set consists of test facts and a number of test cases. A test set represents the description of one situation and a number of queries that is to be asked about this situation.

Definition 3 (Test Set) *A tuple $T_s = (T_f, S_{tc})$, where T_f are test facts and $S_{tc} = \{T_{c1}, ..., T_{cn}\}$ is a set of test cases. The sets of test cases S_{tc} are disjoint, every test case is part of at most one test set.*

A simple example test set would combine just the test facts T_{f1} and the test case T_{c1} defined above, i.e. $T_{s1} = (T_{f1}, \{T_{c1}\})$.

The success or failure of a test case is determined by computing the result for the query using the rule base and the facts defined in the same test set. The results are then compared to the expected result.

Definition 4 (Test Case Success) *A test case $T_{ci} = (q_i, R_i, l)$ that is part of a test set $T_{cj} = (T_{fj}, S_{tcj})$ with $T_{ci} \in S_{tcj}$ is said to succeed iff for all $\theta_x \in R_i$ it holds that $q_i\theta_x \in HM_p(P \cup T_{fj})$, where P is the rule base being tested and HM_p is its perfect herbrand model as defined in section 2.1.4. If R_i is empty, T_{ci} succeeds iff $q_i \in HM_p(P \cup T_{fj})$. Otherwise it is said to fail.*

In the example, the union of the test facts and the program P gives a logic program consisting of

$parent(Y) \leftarrow fatherOf(Y, X)$ # Rule parent 1
$fatherOf(A, X) \leftarrow male(A), child(A, X)$ # Rule fatherOf
$male(peter)$
$child(peter, mike)$

For this logic program the query $q_1 = \{parent(Y)\}$ returns $\theta_1 = \{Y/Peter\}$, i.e. $q_1\theta_1 \in HM_p$ (where HM_p is the perfect Herbrand model of P) and hence the test case succeeds.

A set of tests that a rule base is supposed to pass is called a test suite.

Definition 5 (Test Suite) *A set $S_{ts} = \{T_{c1}, ..., T_{cn}\}$, where $T_{s1}, ..., T_{sn}$ are the test sets for a logic program.*

4.2 Test Coverage

Having a test suite for a program begs the question how good this test suite is; how confident the developer can be that the test set is able to detect most or all faults in the rule base. Test coverage measures are an attempt to answer this question based on how well the test suite executes all parts of the rule base. This section introduces two such measures for normal logic programs. The first - Simple Rule Coverage - just checks whether every rule was involved in answering at least one test case. The second - Complete Rule Coverage - tests whether every rule has been tested in its entire variability.

In order to define such test coverage measures, first a notion is needed of what parts of a rule base were executed to derive a result. This notion - Prooftree - is defined in the next section. Based on this notion, *Usage Data* is defined as the sum of all the data about the execution of all test cases. Rule Coverage and Complete Rule Coverage are then defined based on this notion in the two subsequent chapters.

4.2.1 Prooftree

A **prooftree** is a directed graph $G(A_x, P) = (V, E)$ that exists for each ground atom A_x contained in the perfect Herbrand model HM_p of a logic program P. The prooftree is a declarative representation of the reason for the inclusion of A_x in HM_p.

We first define the justification for the inclusion of an atom A_x in HM_p.

Definition 6 (Atom Justification) *The justification for an atom $A_x \in HM_p$, where HM_p is the perfect Herbrand model of a logic program P, is defined as:*

- *If there is a substitution θ such that $A_x\theta \in P$ then $A_x\theta$ is the justification.*
- *If $A_x \notin P$ then there must be a ground substitution θ and at least one rule $R \in P$ such that A_x is in the head of $R\theta$ and all atoms in positive literals in the rule body are in the Herbrand model. All such instances $R\theta$ are justifications for A_x. Note that this means that there can be multiple justifications for one atom.*

Taking the rule base and the facts from the example in the previous section as an example, the justification for the atom $fatherOf(peter, mike)$ would be given by the rule instance $fatherOf(peter, mike) \leftarrow male(peter), child(peter, mike)$.

Definition 7 (Prooftree) *A prooftree $G(A_x, P) = (V, E)$ for the atom $A_x \in HM_p$ is defined recursively as:*

- *The justifications for A_x are in V.*
- *For each rule instance $R_i = A_x \leftarrow L_1, ..., L_n$ in V, the justifications J_i for each atom in a positive literal in the rule body of R_i are also included in V and edges $e = (R_i, J_i)$ are in E.*

We also define a (partial) function $f_p : V \rightarrow P$ that returns the rule for each rule instance in V.

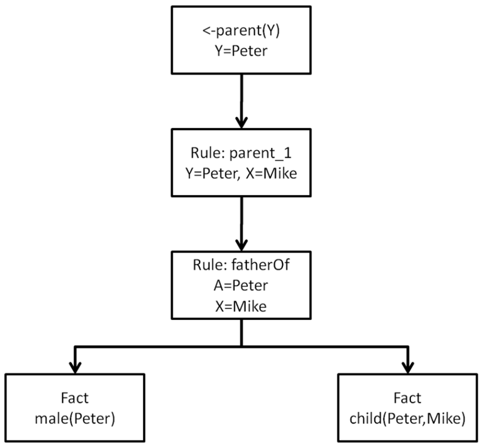


Figure 4.1: *Prooftree for the example rule base*

The prooftree for the example rule base and the query result $parent(peter)$ is shown in figure 4.1. At the top it shows the atom parent(Y) that is returned by the query. The justification for this atom is given by the rule instance $parent(peter) \leftarrow fatherOf(peter, mike)$; because $parent(peter)$ is the head of this rule instance, an edge between these nodes is included. Again the body atom of this rule instance is not in P and hence a rule instance of the $fatherOf$ rule is included as well. Finally the body atoms of this rule instance, $male(peter)$ and $child(peter, mike)$ form the leafs of the prooftree.

Note that in some rule bases there can be multiple justifications for the same atom. This means that at any point the prooftree can have redundant branchens and even that a prooftree can have multiple root nodes of disjunct trees.

4.2.2 Simple Rule Coverage

The prooftree then is a declarative notion of the parts of the rule base that were used to derive a result. Based on this, *test usage data* can be defined as a notion for the parts of the rule base used by an entire test suite.

Definition 8 (Test Case Usage Data) *Test Case Usage Data TU_{tc} for a test case $T_C = (q, R, l)$ that is part of a test set $T_s = (T_f, S_{tc})$ is a set of prooftrees. It consists of all prooftrees $G_x(a_x\theta, P \cup T_f)$ for all atoms $a_x\theta_x$ that can be formed from the atoms of the query $a_x \in q$ and substitutions $\theta \in R$; or the prooftree $G_x(q, P \cup T_f)$ when R is empty and l is true.*

Intuitively, it can be said that test case usage data is the set of all prooftrees for all results to the query. Based on this, Test Suite Usage Data can be defined.

Definition 9 (Test Suite Usage Data) *Test suite usage data $TU_{ts}(T_s)$ is the set of all prooftrees in the test case usage data for all test cases in all test sets of the test suite.*

This defines the set of all prooftrees for all results of all test cases in the test suite. Based on this, 'Simple Rule Coverage' can be defined as a simple notion of test coverage. Intuitively, it captures the notion that a rule has been used in at least one test to the rule base.

Definition 10 (Simply Covered) *A rule $r_x \in P$ is said to be simply covered by a test suite T_s if there is a prooftree $G_x = (V_x, E_x) \in TU_{ts}(T_s)$ in the test suite usage data that contains a rule instance of this rule, i.e. there is a $r_i \in V_x$ such that $f_p(r_i) = r_x$.*

In the example the one prooftree from the one test case contains all rules from the rule base and hence both are simply covered. We can then define Simple Rule Coverage as the proportion of simply covered rules to those that are not.

Definition 11 (Simple Rule Coverage) *If S_c is the set of all simply covered rules in P, then the Simple Rule Coverage f_{sc} is defined as $\frac{|S_c|}{|P|}$*

This notion of test coverage is, however, a very loose measure; a test suite with complete test coverage as calculated by this measure would still not detect many faults. For example replacing the rules in the example by the rule instances from the one test case would mean that the rule base is only useful for exactly these facts - but it would not be detected by the one test case that already provides complete coverage.

4.2.3 Complete Rule Coverage

A more stringent test coverage measure will be defined in the following. It is based on the concept of least general generalizations (lggs) [132] and the notion that a rule is covered, iff the least general generalization of all rule instantiations in the test suite usage data is a variant of the rule itself.

Least general generalization can be understood as the opposite of unification[2]; instead of finding the most general specialization to apply some knowledge in a specific situation, it tries to find the least general generalization that still allows to capture multiple situations. For example the lgg for the atoms $fatherOf(name(mike, lucien), name(michelle, lucien))$ and $fatherOf(name(mike, lucien), name(petra, lucien))$ is $fatherOf(name(mike, lucien), name(A, lucien))$, where A is a newly introduced variable representing the lgg of $michelle$ and $petra$. Within the rules of normal logic programs this is the least general single atom that is at least as general as the two statements given; that can still be unified with these two statements.

The lgg is defined bottom up from terms:

Definition 12 (Least General Generalization for Terms) *The lgg of two terms is defined as:*

$lgg(t_1, t_2) \equiv t_1$*, where* $t_1 = t_2$
$lgg(t_1, t_2) \equiv t_1$*, if only* t_1 *is a variable*
$lgg(t_1, t_2) \equiv t_2$*, if only* t_2 *is a variable*
$lgg(f(t_1, ..., t_n), f(u_1, ..., u_n)) \equiv f(lgg(t_1, u_1)...lgg(t_n, u_n))$
$lgg(t, u)$ *and* $t \neq u \equiv Y$*, where* Y *is the variable that represents* $lgg(t, u)$*.*

Note that a particular variable *represents* an lgg, this means that a variable is used whenever the lgg it is representing appears; e.g. the variable X is always used to represent $lgg(f(a), g(a))$, even if it appears in mulitple literals.

[2]In fact it is also often called anti-unification.

Definition 13 (Least General Generalization for Literals) *The lgg of two literals is defined as:*

$$lgg(p(t_1, ..., t_n), p(u_1, ...u_n)) \equiv p(lgg(t_1, u_1), ..., lgg(t_n, u_n))$$

The lgg is not defined in cases where the predicate symbol is different, or one of the literals is positive and the other negative.

Finally, understanding rule bodies as lists of literals[3], the lgg for rule instantiations can be defined as:

Definition 14 (Least General Generalization for Rules Instantiations) *The lgg of two rule instantiations $r_1 = [l_1, ..., l_n]$ and $r_2 = [m_1, ...m_n]$ of the same rule is then:*

$$lgg([l_1, ..., l_n][m_1, ...m_n]) \equiv [lgg(l_1, m_1), ..., lgg(l_n, m_n)]$$

The lgg for a set of rule instantiations is given by:

$$lgg(\{r_1, ..., r_n\}) \equiv lgg(r_1, lgg(r_2, ...lgg(r_{n-1}, r_n)...))$$

Based on this, a rule is completely covered, iff the lgg of all its rule instantiations is equal to the rule itself (up to the renaming of variables); i.e. it is a variant of the rule.

Definition 15 (Completely Covered) *A rule $r_x \in P$ is said to be completely covered $f_{sp}(r_x)$ by a test suite, iff the lgg of all rule instantiations r_{x-lgg} in all prooftrees of the test suite usage data is a variant of the rule, i.e. there is a substitution θ such that $r_{x-lgg}\theta = r_x$ and a substitution ρ such that $r_{x-lgg} = r_x\rho$*

The complete coverage of the rule base is then defined analogous to Simple Rule Coverage:

Definition 16 (Complete Rule Coverage) *If C_c is the set of all rules in P that are completely covered by a test suite, then the Complete Rule Coverage f_{cc} is defined as $\frac{|C_c|}{|P|}$*

An example shall further clarify this. Starting from the example in the

[3]The ordering of the literals in a rule is not important for the inference process, but choosing an arbitrary ordering for each rule and using the same one for all instantiations of these rules allows for a simpler definition and faster computation of the lgg.

previous section another test set consisting of test facts $male(peter)$ and $child(peter, michelle)$ and again the query $\leftarrow parent(X)$ is introduced. For this example only the rule $parent1$ considered. The rule instantiations for this rule from the two test cases are then:

$$parent(peter) \leftarrow fatherOf(peter, mike)$$
$$parent(peter) \leftarrow fatherOf(peter, michelle)$$

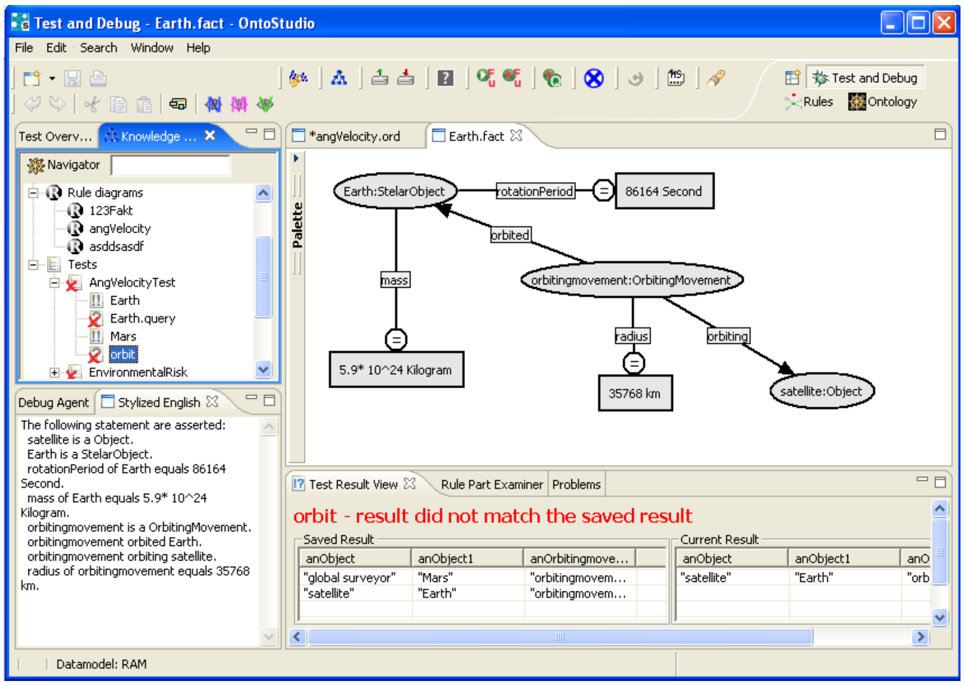
According to the definition of the lgg for literals defined above the lgg of the two body atoms is then given by $fatherOf(lgg(peter, peter), lgg(mike, michelle))$. The definition of least general generalizations for terms then gives $lgg(peter, peter) = peter$ and $lgg(mike, michelle) = X$, where X is a variable representing $lgg(mike, michelle)$. The lgg for the rule $parent_1$ then is $parent(peter) \leftarrow fatherOf(peter, X)$. This lgg is less general than the rule itself, because it includes $peter$ instead of a variable; the test suite for this rule base does not test the full flexibility of this rule - it is only tested for the entity $peter$ as father. Because the lgg is less general than the rule, $parent_1$ is not completely covered. A simple change to one of the test sets that replaces peter with some other identifier, however, would change the rule instantiations in a way that $parent_1$ is completely covered.

4.3 Implementation

An implementation of a testing framework that realizes these test concepts has been done in Project Halo (see section 3.1.2). The focus of this implementation has been on good usability in order to allow domain experts to successfully use it. The testing framework is implemented as a plugin for Eclipse [7] that works within DarkMatterStudio. For the user interface it relies exclusively on the Simple Widget Toolkit SWT [161]. It is implemented following the well known model-view-controller [95] pattern. The different parts of this framework (test fact editor, test query editor, result view ..) are implemented as independent views and editors that do not know of each other, but that share one model and controller.

The testing implementation consists of six main parts. Each of the parts is described in detail in subsequent sections.

- The **test manager** for managing and executing tests. It is responsible for adding the test facts to the knowledge base prior to the execution of tests, computing the result of queries and comparing the result to the intended result. It is not directly visible to the user and includes the model and the controller for the testing framework.



The following statement are asserted:
satellite is a Object.
Earth is a StelarObject.
rotationPeriod of Earth equals 86164 Second.
mass of Earth equals 5.9* 10^24 Kilogram.
orbitingmovement is a OrbitingMovement.
orbitingmovement orbited Earth.
orbitingmovement orbiting satellite.
radius of orbitingmovement equals 35768 km.

orbit - result did not match the saved result

Saved Result

anObject	anObject1	anOrbitingmove...
"global surveyor"	"Mars"	"orbitingmovem...
"satellite"	"Earth"	"orbitingmovem...

Current Result

anObject	anObject1	anO
"satellite"	"Earth"	"orb

Figure 4.2: *Overall test interface, showing the stylized english view at the bottom left, the test fact interface in the top right and the result view at the bottom right.*

- The **test overview** - an interface that allows users to manage tests and to get a quick overview about which test succeeded and which failed.
- The **test fact interface** allows users to create test facts in a simple, graphical way.
- The **test query interface** allows users to create test cases in a simple, graphical way.
- The **stylized English view** that shows a textual representation of the current query or fact diagram.
- The **result view** that shows the current results and allows to specify the expected result.

The combination of the different parts of the interface is shown in figure 4.2. At the top left is the navigation interface that shows different elements from the knowledge base, in this location either the overall knowledge navigator or the test overview is shown. Below that, on the lower left hand side the stylized English view is shown. The biggest part of the interface is taken by either test fact or test query Interface - depending on which is currently opened. Below these editors the result for the currently selected test query is shown.

4.3.1 Test Manager

The test manager realizes the test concepts introduced in the beginning of this chapter. It is responsible for managing the test suite associated with the working environment of one developer. The test suite consists of test sets, each of which contains test facts and a number of queries. In addition to these concepts, the test manager uses the notion of 'diagram'; the graphical representation of a query or a number of facts that is edited by the developer. One query is always represented by one query diagram, the test facts of one test set, however, can be spread over multiple fact diagrams. An overview of the model is shown as UML class diagram in figure 4.3.

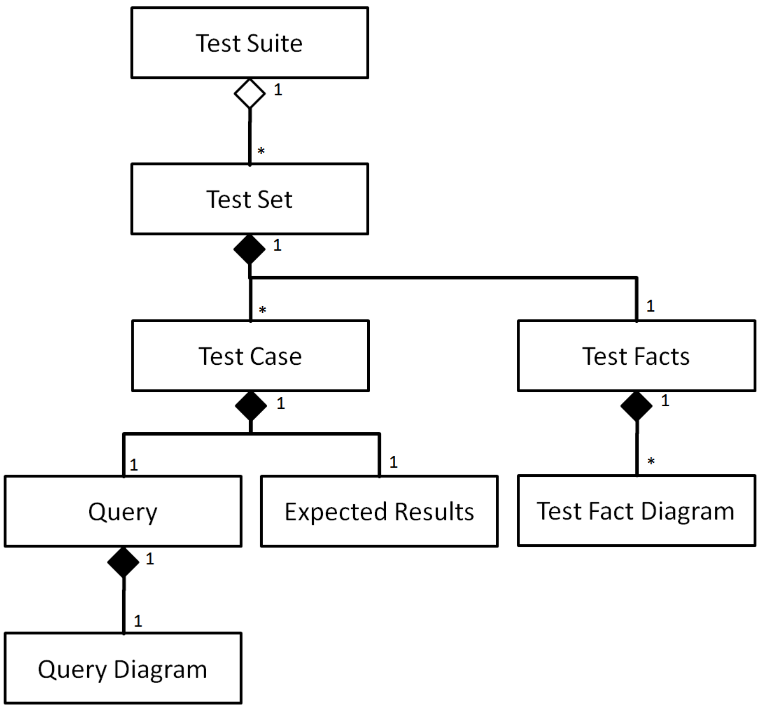


Figure 4.3: *Simplified test framework implementation model*

Not shown in this simplified object model are the relations to other parts of Dark Matter Studio. Of particular importance are rule test sets - a specific kind of test set that is directly associated with a rule. These test sets get created for a particular rule and stay connected to this rule for its entire lifecycle. The main difference to normal test sets is, that when a rule gets deactivated or removed from the rule base the test is deactivated or removed as well. This was an important feature since users of the system often deactivated rules during development of the rule base - causing a large number of unnecessary test failures.

Another extension to the model presented at the beginning of the chapter is a slightly more complex notion of test case success. This was necessary because users were often unwilling to specify the expected result for test cases. Instead of specifying the expected result, they viewed getting *any* result to a query as the success of a test case. The test manager was hence extended to understand three possible states of test cases:

- test case success: an expected result is specified and matched by the current result.
- probable test case success: no expected result is specified, but the test case query returns some result.
- test case failure: an expected result is specified and not matched by the current result or no result is specified and the query returns no result at all.

4.3.2 Test Overview

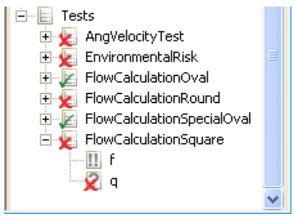


Figure 4.4: *Test overview interface*

The test overview interface is used to add, remove, open and, run; i.e. to manage test sets, test case and test facts. This view shows all test and with one glance allows to see which succeed and which fail. In the interface (see figure 4.4) test query diagrams are shown as documents with small question marks, test fact diagrams with two exclamation marks. Red and green decorations are used to mark test cases that fail or succeed. A probable test case success is shown by the absence of any mark. The test overview allows creating new test entities by clicking on the embedded toolbar or through a context menu.

Finally the test overview also allows user to start any number of test cases with one click. A progress dialog is included that shows the progress of the tests, allows to cancel the test execution and that shows a short summary of the results of running the tests.

4.3.3 Test Fact Interface

Test facts and queries are created using variants of the Dark Matter Studio rule editor. Besides the obvious advantage of saving implementation effort, this also helps the domain experts because they only have to learn to use one editor. The reuse of concepts from the Rule Editor is easily possible because in the end a fact is just a rule without a body and a query is a rule without a head. We will start by explaining the editor in its role as test fact editor.

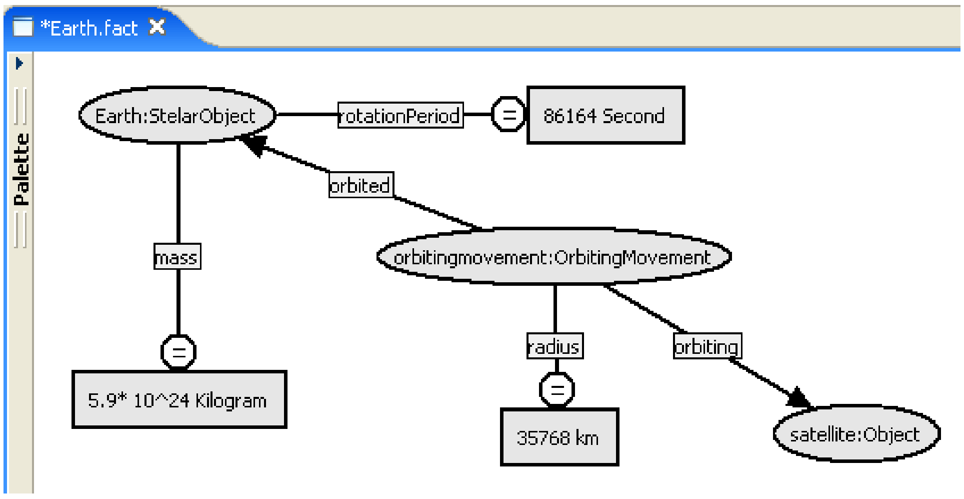


Figure 4.5: *An example fact diagram*

One example fact diagram is shown in figure 4.5. This example realizes test facts of the kind that where needed for the physics domain in the Halo project. This example diagram describes earth, its mass and rotation period, and a satellite orbiting the earth at some radius. This could be a test setup for tests that check whether the rules that identify and calculate a geostationary orbit are correct.

The components of a fact diagram are instance nodes, predicates, attribute values, attributes and relations. Instance nodes represent F-logic statements of the form“Instance:Class”. Instance nodes are displayed as ellipses. “Earth is a Stellar Object” is an example of such a node in Figure 4.5. The user can create an instance node by selecting the appropriate tool from the context menu or the palette in the editor and then choosing an existing concept and a name for the instance. Alternatively, a concept or an instance can be dragged from the concept hierarchy in the knowledge navigator into the editor. Instance nodes are connected by relations. The possible relation types are restricted by the schema information about concepts of the instance nodes connected by a relation link. Relation links that are not consistent with the schema are flagged as problematic, highlighted in orange and reported in a problems view. Attribute values are shown as rectangles and connected to the instance nodes by attribute links. Here, too, the editor ensures that the attributes correspond to the schema.

Note that - in the interest of usability - this fact editor is restricted in expressiveness; in particular it does not offer a way to enter any function symbols or subclass statements. I.e. the fact editor is restricted to creating instances and adding data-atoms for these instances.

4.3.4 Test Query Interface

The actual test queries are created with a variant of the same rule editor. It allows the user to specify the intended result as well as to create queries that contain variables. In the first case, the test query will check whether the result can be inferred, in the latter case the result will be displayed and verified by the user.

When the editor is used to specify an intended result, it is used exactly like the fact editor: The diagram consists of instance nodes, attribute values and predicates that are connected with relations, attributes and links indicating the usage of an instance in a predicate. Only that, because the graph as a whole is specified as a query, the system knows that these facts should not be added to the knowledge base, but rather checked whether they can be inferred.

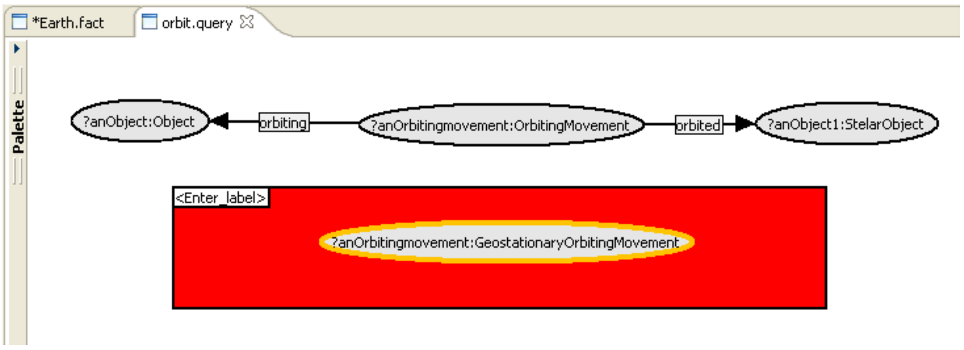


Figure 4.6: *The test query interface*

An example of the query editor used to specify a query with variables is shown in Figure 4.6. The query asks for all instances of OrbittingMovement, the StellarObject that is orbited and the object that is orbiting. The results that are returned are restricted so that they do not include those orbiting movements that are instances of GeostationaryOrbitingMovement.

In short: This example query asks for information about orbitting things that are not in geostationary orbit.

The main building blocks of the query editor are still the instance nodes, only that now it is allowed to substitute variables for the instance names. The same holds for attribute values that can be replaced by variables. A query diagram can obviously combine instance nodes with variables and instance nodes referencing instances in the knowledge base in any way.

The test query editor allows for negated query parts. Negation of parts of the query is represented by the concept of *exception*. An exception is drawn as a red box in the query diagram. Query entities drawn in this box are negated. For instance, the instance node inside the large red box in the example in Figure 4.6 appends the literal "not anOrbittingmovemen:GeostationaryOrbitingMovement." to the query.

In the interest of usability, the editor's expressivity is restricted relative to what is possible in F-logic. In particular, the editor does not allow disjunctive queries, variables for concepts (e.g. "return all classes for the instance Earth") or variables for relations (e.g. "return all relations between Mars and Global Surveyor"). Allowing variables for relations or concepts would have made it impossible to do schema-based validation and error reporting of the entered data. Disjunction would have made the diagrams very hard to read.

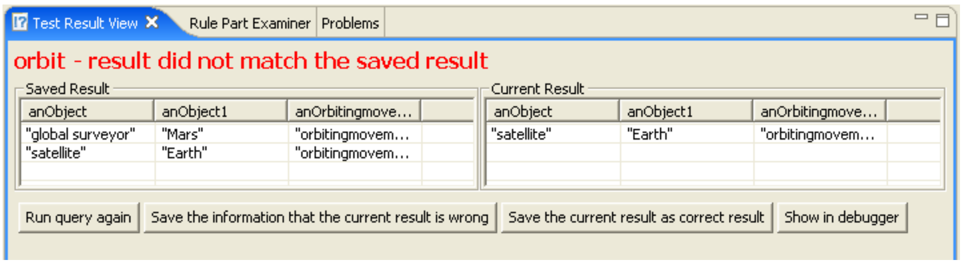
4.3.5 Stylized English View

The content of fact and query diagrams is also displayed in a form of a stylized English explanation text; this should help prevent mistakes due to a misunderstanding of the graphical language of the editors. The text shown in the stylized English view changes automatically to immediately reflect the changes the user makes to the diagram. The automatically generated stylized English text for the example in Figure 4.5 reads:

```
The following statement are asserted:
satellite is a Object.
Earth is a StellarObject.
rotationPeriod of Earth equals 86164 Second.
mass of Earth equals 5.9e24 Kilogram.
orbitingmovement is a OrbitingMovement.
orbitingmovement orbited Earth.
orbitingmovement orbiting satellite.
radius of orbitingmovement equals 35768km
```

4.3.6 Result View

The results of a query are displayed in the test result view shown in Figure 4.7. It shows the result that was saved as correct and the current result side by side. To the left it shows the saved result as table. The table has one column for each variable in the query and one row for each set of variable bindings that satisfy the query. The example in the screenshot in Figure 4.7 shows the saved result for the query presented in Figure 4.6. A different table to the right shows the result from the last run of the test. In the example, the current result is only a subset of the saved result and hence the query fails. Buttons below the result tables allow to run the query again, designate the current result as correct, to designate it as false and to open the current test in the debugger. The test result view does not offer any direct way to input the intended result, the user can only designate a current result as correct or false. Users that want to directly define the intended result can use the query editor described above to describe it in a query without variables. The result view then changes to display only whether this intended result can be inferred or not.


Test Result View
Rule Part Examiner
Problems
Run query again
Save the information that the current result is wrong
Save the current result as correct result
Show in debugger

orbit - result did not match the saved result

Saved Result

anObject	anObject1	anOrbitingmove...
"global surveyor"	"Mars"	"orbitingmovem...
"satellite"	"Earth"	"orbitingmovem...

Current Result

anObject	anObject1	anOrbitingmove...
"satellite"	"Earth"	"orbitingmovem...

Figure 4.7: *Result view*

4.4 Evaluation

Most of the design elements described in the previous section emerged in a process of iterative comparison and testing of alternatives; in fact two whole interfaces were discarded before the presented design for test and query editor was developed. The first of these interfaces was text based and too difficult to use for end users. The second was a graphical editor but it used symbols and concepts different from the rule editor; hence it required domain experts to learn another graphical language - something that is not needed with the third and current implementation.

Further improvements in detail are certainly possible; but the important question to evaluate was: is this general approach to testing sufficiently natural and manageable for domain experts so that they can apply it effectively over extended periods of time to large and complex bodies of knowledge?

Method

The evaluation of the testing framework was performed as part of the overall evaluation in the second phase of Project Halo, already described in section 3.1.2. For this evaluation most relevant is the 6-week knowledge formulation phase, during which the domain experts worked 4–5 hours each day. The knowledge formulation phase began with 40 hours of training with Dark Matter Studio, about three-quarters of which was devoted to supervised practice. Then each domain expert spent 100 hours independently using Dark Matter Studio to formalize knowledge from two chapters of a college-level textbook from his or her domain. The main topics in physics, biology, and chemistry were kinematics, cell structure and processes, and stoichiometry[4], respectively.

Each domain expert either had already earned a master's degree in the domain in question or was taking master's-level courses. None had experience with knowledge engineering; two had some programming experience, but it was not related to the type of formalization required with Dark Matter Studio.

Use of the Test and Debug Component

All domain experts participating in the evaluation used the test component extensively. On average more than 30% of the time was spent in the T&D (Test and Debug) component that also included a debugging tool. This high proportion of time spent in this component is, however, not exclusively a consequence of the importance of testing, but also caused by the long time that running some of the tests took[5].

[4]The quantitative study of reactants and products of chemical reactions.

[5]A consequence of the overall runtime problems faced in Project Halo - see section 3.1.2.

	Time spent in DMS (hh:mm)	Time spent in T&D (hh:mm)	% of time spent in T&D
DE 1 (Physics)	106:56	29:37	27.7%
DE 2 (Biology)	81:13	22:30	27.7%
DE 3 (Biology)	128:25	59:42	46.6%
DE 4 (Chemistry)	96:24	35:26	36.8%
DE 5 (Chemistry)	84:32	17:08	20.3%
DE 6 (Physics)	102:54	28:55	28.1 %

Table 4.1: *Use of test and debug component for each domain expert*

During this time an average of 858 test cases where evaluated, of which almost 81% ran successfully and almost 19% were either aborted by the user or ended in an exception in the inference engine.

	# of test runs	# of successful test runs	# of test runs with exceptions or aborts
DE 1 (Physics)	1194	1001	193
DE 2 (Biology)	492	433	59
DE 3 (Biology)	601	448	153
DE 4 (Chemistry)	861	682	179
DE 5 (Chemistry)	901	770	131
DE 6 (Physics)	1097	883	264

Table 4.2: *Number of test runs for each domain expert*

The thoroughness with which the domain experts tested the rules varied widely - even though all domain experts were asked to test extensively. In the end some domain experts had tested barely 30% of their rules while others had more tests than rules.

	# of rules	# of tests	% of enabled rules tested
DE 1 (Physics)	50	24	51%
DE 2 (Biology)	69	56	93%
DE 3 (Biology)	130	57	46%
DE 4 (Chemistry)	146	36	29%
DE 5 (Chemistry)	175	49	28%
DE 6 (Physics)	40	48	115%

Table 4.3: *Number of rules and tests for each domain expert*

It can be seen that the physics domain experts were able to get by with a smaller number of rules (reflecting the fact that much of the knowledge in the physics syllabus could be captured by some key equations), which they were able to test more successfully. In the other two domains, the domain experts found it necessary to define considerably larger numbers of rules, but they were able to test only roughly the same absolute numbers of rules successfully.

The poorer results for the testing of these larger rule sets can be explained in part by technical problems of the inference engine: whereas the execution times with the small rule sets of physics were quite acceptable (on the order of a few seconds), in the other two domains the domain experts often had to wait several minutes or more, and they accordingly often aborted tests. Moreover, these large rule sets gave rise to forms of interference among rules that had not been encountered in the previous smaller-scale tests or in the work of experienced knowledge engineers with the system. By taxing the patience of the biology and chemistry domain experts, these problems offer an opportunity to see how important these domain experts considered the activity of testing. We have already seen that these domain experts spent a comparable amount of time using the T&D perspective and that they initiated hundreds of test runs despite the difficulties just mentioned.

Overall Responses to the Test Component

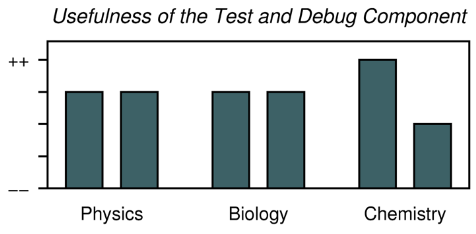


Figure 4.8: *Respones to test and debug component*

A different perspective is given by Figure 4.8: The domain experts were asked in the final interview to express their degree of agreement with the statement “I found [this component] useful in helping me to achieve my goals.” Here the pattern is consistent: All domain experts express moderate or strong agreement with the statement for the test and debug component. Overall for all components, domain experts had by no means a general tendency to express satisfaction with. For example, two components that were intended to facilitate the definition of new elements of the ontology on the basis of occurrence of terms in the text were found to be relatively unimportant: The domain experts felt quite capable of specifying the necessary ontology elements themselves.

Qualitative Observations

Even domain experts with no previous experience in knowledge engineering or programming were able to use the testing framework. They showed a strong inclination to test their rules as they went along, and they found the testing tool provided to be basically suitable for this purpose (except for the performance problems that they experienced). All domain experts were able to create and run a large number of tests.

The later question answering phase and the examination of the created rule bases revealed that only sufficiently tested knowledge bases would later work. The knowledge bases of the two physics SME that were tested most extensively were also the ones most successful at question answering (see section 3.1.2).

One severe drawback of the system during the evaluation was that sometimes running all tests took a very long time; some domain experts even had to run their tests overnight. This lack of immediate feedback and interactivity was one reason for why testing has sometimes been neglected and tests did not work properly. These runtime problems, however, were not caused by the testing component but were rather a consequence of overall issues with the inference engine encountered in this phase of Project Halo.

4.5 Prior Work

Little academic literature exists about the testing of logic programs, even though it is used routinely and is often implemented as part of rule base development environments.

A simple notion of tests is given in [12, 13] for the purpose of defining a test coverage measure. In this paper tests and test input is defined as a set of atomic goals, without any notion of test facts as defined in the beginning of this chapter. Paschke et. al [127] take an approach more similar to the one presented here, defining a test case as a set consisting of temporarily asserted test input facts, one or more test queries and, for each query, an expected answer label and an output result set. This paper also defines a format to represent tests in XML and to include them in RuleML [141] files. This definition, which postdates the one presented here, uses a different terminology but is conceptually compatible.

A different approach is taken by Ruggieri [140], he defines a test as finite set of arbitrary, possible non-ground, atoms. The set represents requirements to the program based on requirements documents, the specification or previous versions of the program. Testing then becomes the checking whether the formal semantics of the program entails this set of atoms. In this model, however, testing is not generally decidable.

Test coverage metrics for logic programming based on least general generalizations have been proposed before by Belli and Jack [13] and by Paschke et. al [127]. Both approaches, however, only calculate the lgg for the specialization given by the unification of the test queries with rules. The approaches do not consider any actual results of the query or the inference process beyond the unification of the query with the top-level rule. Because a rule has to unify directly with a test query in order to be covered, an unnecessarily large number of tests is required. As an example again consider the rule base below and the query $q_1 = \{parent(Y)\}$.

$$\begin{aligned}
&parent(Y) \leftarrow fatherOf(Y, X) \text{ \# Rule parent 1}\\
&fatherOf(A, X) \leftarrow male(A), child(A, X) \text{ \# Rule fatherOf}\\
&male(peter)\\
&child(peter, mike)
\end{aligned}$$

In order to process this query, the inference engine uses both rules and hence the test coverage measure proposed in this chapter will find both rules to be partially covered. The test coverage metrics proposed by Paschke and Belli, however, only consider the rule that can be directly unified with the query, i.e. in this example only the rule $parent1$; these test coverage metrics will still show the rule $fatherOf$ to be not covered at all, hence will underestimate the actual coverage of the rule base.

Another problem for the two approaches concerns *trivial covers*. A trivial cover for a rule base can be created by taking all rule heads as test queries. None of the queries may return any result, but the rule base is fully covered[6]. Jack and Belli introduce restrictions on the queries to obtain non-trivial covers—a complication that is not needed in the approach presented here. As an example consider the rule base (the same as above, only the facts have been removed):

$$\begin{aligned}
&parent(Y) \leftarrow fatherOf(Y, X) \text{ \# Rule parent 1}\\
&fatherOf(A, X) \leftarrow male(A), child(A, X) \text{ \# Rule fatherOf}
\end{aligned}$$

A trivial cover for this rule base can be obtained with the queries $q_1 = \{parent(Y)\}$ and $q_1 = \{fatherOf(A, X)\}$. The least general generalisation of the unification of these queries with rule $parent1$ and rule $fatherOf$ are variants of the rules and hence Paschke's and Belli's[7] measures will show both rules to be fully covered. This is problematic, because these tests

[6]These covers are called trivial, because in the case where no result is expected or result they do not add *any* information to the rule base.

[7]Without the complicating restrictions.

do not add any information to the rule base; because these test coverage measures count nonsensical tests. With these measures a test suite giving a complete test coverage can be created fully automatically for all rule base that do not contain facts. The approach proposed by the author does not suffer from this weakness, because it considers not the unification of the query with the rules but the actual rule instantiations used to derive results.

A different approach is taken in the TRUBAC[8] [6, 5] system. Here the rule base is translated into a graph reflecting the possibilities of rules to interact and to use facts. Different test coverage measures are then defined based on how well different classes of paths are used in the evaluation of the test cases. A similar approach is taken by Kleiner [94] who also defines a visualization of the coverage of a rule base. Compared to the approach introduced here, these approaches are stronger and require a larger number of tests to achieve complete test coverage; in fact the number of tests needed is so large, that complete test coverage becomes unattainable for all but the simplest rule bases.

4.6 Conclusions

This chapter presented three different contributions

- A formal account of the notions of test set, test case, test facts and test suite that transfers these important notions to normal logic programs.
- The definition of a novel test coverage measure based on least general generalization that avoids the problems of earlier measures that underestimating test coverage in some cases and overestimate it through trivial covers in others.
- The implementation and evaluation of a test framework based on these notions.

The evaluation showed that these testing concepts as well as the testing interfaces were usable by domain experts. Even domain experts who had previously mainly used standard office software quickly caught on to tools for testing an ontology and associated rules, all were able to create and run a large number of tests. This in turn helped them improve their rule development skills and to create working rule bases.

Qualitative evidence from observations and interviews of the domain experts further showed the large interest and importance domain experts placed on testing; testing was very important for their motivation and for them to learn about the knowledge engineering.

[8]Testing with Rule Base Coverage.

The presented solution is the first graphical testing framework for F-logic and it tackles the Problem of Tool Support for the dynamic analysis transition in the fault lifecycle (see section 3.2). It is further specifically tailored to better support end user developers and well suited to support agile development of rule bases. The presented testing solution realizes the Interactivity principle (by enabling developers to quickly try out the developed rule base) and the declarativity principle (none of the proposed concepts depend on the procedural nature of the inference engine).

Chapter 5

Debugging

Debugging is the process of tracking down and correcting faults based on errors, differences between an observed or measured value or condition and the true, specified or theoretically correct value or condition [84]. Debugging is a two step process, first a diagnostic step that identifies the fault that caused the observed bug and second, changes to the rule base to rectify the fault. In most development environments debugging is supported by dedicated tools that can greatly speed up the debugging process. Most of these tools center on the diagnostic step, relying on the overall creation and editing tools for the second step.

Debugging is an inevitable and important part of any software development activity, and good debugging support is an important ingredient for the efficient creation of software. These statements are equally true for all programming paradigms; debugging may look different for rule based systems, but it is just as needed as it is for assembler programming.

The importance of debugging was further emphasized by the survey of developers presented in section 3.1.1. Asked to compare rule base development to the development with procedural and object oriented programming languages, the majority of participants found that rule based systems development processes are inferior with respect to ease of debugging. In a later question the difficulty of debugging was also given most often as the issue hindering the development of rule bases.

In the analysis (section 3.2) five issues were identified as most affecting the debugging of rule bases

1. **The Problem of Interconnection** is hampering fault localisation because it leads to errors appearing in surprising places.

2. **The no-result-case and the Problem of Error Reporting** make it hard

to diagnose faults, because the error in many cases simply is an empty result set.

3. **The Problem of Procedural Debugging** forces developers using state of the art debugging support to learn about the procedural nature of the inference engine; thereby brakes the declarative paradigm and forces developers to learn about the *how* of computation.

4. **Embedded Rule Bases and Embedded Procedures** that lead to hard to diagnose faults at the boundaries of different programming paradigms, but also mean that rule bases tend to be relatively small.

5. **The Emerging Semantic Web and Rule Interchange** that mean debugging may include rule bases developed independently, without central control; meaning that often the developer doing the debugging has to learn about the rule base at the same time.

The design principles of **interactivity**, **visibility** of the hidden rule base structure, **declarativity** and **modularization** were identified (see section 3.3) as additional requirements for tools addressing the debugging challenge.

This chapter presents Explorative Debugging as a new debugging paradigm for rule based systems. It concentrates on the issues identified in the analysis and realizes the design principles proposed. Explorative Debugging tackles the Problem of Procedural Debugging by making it possible to explore the execution of a rule base at the declarative level. The Problem of Interconnection and the Problem of Error Reporting in the no-result case are tackled with Partial- and Mutated Prooftrees introduced in this chapter (section 5.1.3). The proposed debugging paradigm enables developers to explore and quickly do experimental changes to the rule base - realizing the interactivity design principle and enabling users to learn about unfamiliar rule bases during debugging. Overall the visibility design principle is realized through a multitude of (partially novel) techniques to show the static (section 5.1.2) and dynamic structure (section 5.1.3) of the rule base. A specific way to deal with faults at the boundaries to procedural built-ins was also implemented and is presented in section 5.2.2.

The chapter starts with the definition of Explorative Debugging and its components. After that the two implementations of the Explorative Debugging paradigm for RDF rules and F-logic are presented. The F-logic implementation is a more conservative implementation that is integrated into a commercial knowledge engineering workbench while the RDF rules implementation is more powerful but less refined. Also within the implementation section a debugging walkthrough using the F-logic debugger is presented. The section thereafter describes the comparative evaluation of the debugger for RDF rules with a procedural debugger. After that a long section gives an overview of prior work in rule debugging before the chapter concludes.

5.1 Explorative Debugging and its Components

An Explorative Debugger is a rule browser that enables the user to browse through the declarative structure of a rule base and its execution. Explorative Debugging is defined as:

Definition 17 (Explorative Debugging) *Finding faults in a rule base based on a developer-controlled exploration of the inference process that leads to an error. This exploration process is done at the level of the declarative semantics of the rule language, it is independent of the strategy of the inference engine. The debugging space that is explored consists of*

- *The rules themselves, that can be experimentally altered to explore the effects of changes.*
- *The conclusions enabled by rules.*
- *The static structure of the rule base.*
- *The dynamic structure of the inference process that led to the error.*

An Explorative Debugger is a browser like tool that supports this process by showing the different components of the debugging space and allowing to navigate along the connections between the entities in it.

In this section each of these components will be described in more detail. The actual implementation will be the topic of the section after this.

5.1.1 Rules and Rule Firings

The main elements for an Explorative Debugger are rules. The debugger is always focused on one rule and can be opened for any rule. Navigation elements allow navigating between rules. The decision to center the debugger on rules was taken because the user syntax of every rule language known to the author as well as the user interface of most rule editing tools is rule centric as well. Centering a debugger around rules sounds self evident, but is not realized in many existing debuggers for rule based systems (these debuggers use the goal the inference engine currently tries to prove as main element).

An important component is the opportunity to quickly change the rules to further explore the role different parts of the rule play. A simple way to do this is to enable the user to quickly deactivate parts of the rule body to check which effects this has on the rule firings. This is also the approach realized in the implementations described below. Any kind of experimental rule change could be supported, however, an integration with rule editing tools could serve a similar purpose.

Rule firings are the most basic form of checking whether a rule performs according to the expectations of the user, they enable the user to quickly check whether a given rule performs the expected function.

Definition 18 (Rule Firings) *The rule firings of a rule E with respect to a rule base R are all ground instances $E\theta$ in the perfect Herbrand model HM of R.*

For a rule $A \leftarrow B$ (where A is a conjunction of atoms and B is a conjunction of literals) these rule firings can be computed simply from the results to the query $\leftarrow B$.

While very simple from a theoretical point of view, practical application of rule firings can be problematic. On the one hand the number of rule firings can be very large for some rules, requiring methods to summarize large result lists or even needing a way to compute only partial results to ensure the responsiveness of the program. Emphasizing the use of relatively small test cases can also help reduce this problem. On the other hand rule firings cannot be computed at all for some rules that utilize built-ins, e.g. $larger(a, b) \leftarrow greater(a, b)$ where $greater(a, b)$ is built-in. This kind of rules would require other user interface methods to enable the user to check whether they perform to her expectations.

5.1.2 Static Structure

The simplest notion of static structure of a rule base is given by the Depends-On Graph. We define this well known structure (e.g. [33]) in the following way:

Definition 19 (Depends-On Graph) *The depends-on graph of a rule base R, is a directed graph $G_R = (R, A)$ with*

- *The set of vertices R, there is one vertex for each rule in the rule base. We write that R_{E_1} is the vertex for rule E_1.*
- *The set of ordered pairs A of vertices called arcs or arrows. An arc $e = (R_{E_1}, R_{E_2}) \in A$ exists iff at least one body atom of E_1 can be unified with at least one head atom of E_2.*

Intuitively, the depends-on graph captures the notion that one rule concludes a result that is used by another rule; an arc $e = (x, y)$ is considered to mean that the rule represented by the vertex x depends on the rule represented by y.

Depends-on connections computed in this way overestimate the potential of rules to interact; hence approaches exist to impose additional constraints [93, 143, 74]. Usually these approaches require that two rules connected through a depends-on-link are consistent with respect to some predefined criteria. A different approach is taken by [176], it uses actual rule interactions during the use of the rule base to compute the *strength* of a depends-on connection.

An important limitation of the Depends-On Graph is the lack of support for cases where a fault prevents the existence of a depends-on connection. For example a typo affecting a ground term in either the head or the body of a rule may cancel a depends-on connection; a depends-on graph will not offer any support for finding such a fault. To address this problem the mutation-extended (mutex) depends-on graph is defined.

The mutex depends-on graph is defined to give a declarative notion of static dependencies that *almost* exist, static dependencies where a rule body *almost* unifies with another rule's head. A mutex depends-on graph is a directed graph with connections between all rules, where the strength of the connection between two rules is inverse to the size of the change that would need to be performed on the rules in order to have a static dependencies (a depends-on connection as defined above). To capture this notion we first formalize the notion of change as *mutation*. A *mutant* is then defined as the application of a mutation to a rule and *mutation similarity* as a measure for the size of the change. Finally the mutex depends-on graph is defined based on these notions.

Definition 20 (Mutation) *A mutation ρ is a set $\rho = \{t_1/u_1, ..., t_n/u_n, p_1/q_1, ..., p_m/q_m\}$*

- *where $t_1, ..., t_n$ and $u_1, ..., u_n$, are terms and $t_1, ..., t_n$ are distinct*
- *where $p_1, ..., p_m$ and $q_1, ..., q_m$, are predicate symbols and $p_1, ...p_m$ are distinct.*

Based on this a Mutant of a rule is defined as the application of a mutation to a rule.

Definition 21 (Mutant) *We call the rule obtained by applying a mutation $\rho = \{t_1/u_1, ..., t_n/u_n, p_1/q_1, ..., p_m/q_m\}$ to a rule E a* **mutant** *of the rule, this is written as $E\rho$. The mutant is obtained through the simultaneous replacement of all terms t_x in the rule with the corresponding u_x and of predicate symbols p_y with the corresponding q_y.*

Based on this we define the Mutation Similarity as a measure for the size of the change implied by a mutation.

Definition 22 (Mutation Similarity) *The mutation similarity $f_{sim}(\rho)$ is defined as the aggregation of the Term and Predicate Similarity values of all replacements, i.e. $f_{sim}(\rho) = f_{aggregation}\,(f_{simTerm}(t_1, u_1), ..., f_{simTerm}(t_n, u_n), f_{simPred}(p_1, q_1), ..., f_{simPred}(p_m, q_m))$. f_{sim} of the empty mutation $\rho_e = \emptyset$ is defined to be 1.*

The function $f_{aggregation}$ can be any function to aggregate the term similarities; a typical function would be the product or the sum of all term and predicate similarity functions. The selection of the right aggregation function is an empirical question and depends on the rule language and, in particular, the kind of errors that get made.

Definition 23 (Term and Predicate Similarity Functions) *The term similarity function $f_{simTerm}$ and the predicate similarity function $f_{simPred}$ are arbitrary similarity functions, returning values between 0 and 1, with 1 representing identity and 0 maximum dissimilarity.*

A simple term similarity function would be the inverse of the edit distance between the terms. More sophisticated similarity functions could utilize semantic similarity measures (e.g. [103]), such as the taxonomic distance between types or type similarity computed by an ontology mapping approach.

Definition 24 (Mutation Extended Depends-On Graph) *The Mutation Extended Depends-On Graph (Mutex Depends-On Graph) of a rule base R, then, is a directed graph $G_R = (R, A)$ and a function $f_s : A \rightarrow [0, 1]$ with*

- *The set of vertices R, there is one vertex for each rule in the rule base. We write that R_{E_1} is the vertex for rule E_1.*
- *The set of ordered pairs A of vertices called arcs or arrows. An arc exists for every possible pair of rules.*
- *The function f_s representing the strength of the evidence for a particular arc. The value of f_s for an arc $a(R_{E_1}, R_{E_2})$ is defined as the maximum mutation similarity $f_{sim}(\rho)$ for all body atoms $a_{1,x}$ of E_1, all head atoms $a_{2,y}$ of E_2 and all mutations ρ_z such that there exists a substitution θ such that $a_{1,x}\theta = a_{2,y}\rho_z\theta$.*

Intuitively, it can be said, that the mutation extended depends-on graph has connections between all rules, the strength of each arc representing the minimum change that would be needed in order to unify at least one body atom of a rule with the head atom of another. The mutation extended depends-on graph contains all arcs a_x of the normal depends-on graph with $f_s(a_x) = 1$, since per definition of the normal depends-on graph, these arcs represent rule pairs for which a body atom can be unified with a head atom without any change.

Practically, it obviously makes no sense to either compute or show the entire mutation extended depends-on graph. Rather a strict minimum similarity needs to be imposed on the computation and the display of the extended depends on graph. Depending on the similarity function and on the expressiveness of the rule language it might also be necessary to compute the extended depends-on graph only heuristically.

5.1.3 Dynamic Structure

The basic dynamic structure of the rule base is given by the prooftree that in section 4.2.1 was defined as a graph $G(A_x, P) = (V, E)$ for an atom $A_x \in HM_p$:

- The justification for A_x is in V.
- For each rule instance $R_i = A \leftarrow L_1, ..., L_n$ in V, the justification J_i for each atom in a positive literal in the rule body of R_i is also included in V and an edge $e = (R_i, J_i)$ is in E.

Although normally not defined in this declarative way, variants of prooftrees have been used successfully for many years in the explanation of expert systems (e.g. [36, 108, 62]). The use of prooftrees for debugging (and to a lesser extent also for explanations) is, however, limited because of their inability to explain the failure of the rule base to arrive at an expected result; a prooftree is not defined for atoms $A \notin HM$.

As an example consider the following rule base, and the atom $parent(peter)$. This atom is not a valid conclusion of this rule base, it is not in its Herbrand Model and hence there is no prooftree for this atom.

$parent(Y) \leftarrow fatherOf(Y, X)$ # Rule parent 1
$fatherOf(A, X) \leftarrow male(A), child(A, X)$ # Rule fatherOf
$child(peter, mike)$

However, we can see that $parent(peter)$ can *almost* be concluded. Adding just one fact ($male(peter)$) would make $parent(peter)$ a conclusion of this rule base. The partial prooftrees introduced below are a way to capture and communicate this notion of a proof that *almost* succeeded.

5.1.4 Partial Prooftree

A **partial prooftree** is a directed graph $G(A) = (V, E)$ that exists for every ground atom A that can be formed using the terms and predicate symbols in the rule base. To define it, first the notion of partial justification needs to be introduced.

Definition 25 (Partial Atom Justification) *The partial justification for an atom $A_x \in HM_p$ is defined as:*

- *If there is a substitution θ such that $A_x\theta \in P$ then $A_x\theta$ is the justification.*
- *If $A_x \notin P$ then there can be a ground substitution θ and at least one rule $R \in P$ such that A_x is in the head of $R\theta$ and all atoms in positive literals in the rule body are in the Herbrand model. All such instances $R\theta$ are the justifications for A_x.*
- *If there is neither of the above, the justification is the* null justification*.*

Based on this, the partial prooftree can be defined:

Definition 26 (Partial Prooftree) *A Partial Prooftree $G_p(A_x, P) = (V, E)$ for the atom A_x in the logic program P is defined recursively as:*

1. *The partial justification for A_x is in V.*
2. *For each rule instance $R_i = A \leftarrow L_1, ..., L_n$ in V, the partial justification J_i for each atom in a positive literal in the rule body of R_i is also included in V and an edge $e = (R_i, J_i)$ is in E.*

Intuitively, a partial prooftree can be understood as the set of (successful as well as failed) proofs of a top-down inference engine; note, however, that its definition does not depend on any particular evaluation strategy of the inference engine. Partial prooftrees are augmented by a scoring function f_{pp} that computes the grounding of a prooftree in the rule base, for example through the proportion of justifications that are null justifications.

Computing all partial prooftrees defined in this way is an intractable problem, the partial prooftrees would be very confusing to the user and very often they would be of infinite size. For this reason, any practical application must minimize the size of the partial prooftrees while maximizing the value of the scoring function f_{pp}. One algorithm to do this is described in the next section.

Coming back to the example from the beginning of this section, it can be seen that there is a partial prooftree for the atom $parent(peter)$ that includes the null justification for the atom $male(peter)$ (see figure 5.1). Note that there are also partial prooftrees for all imaginable atoms; for example the

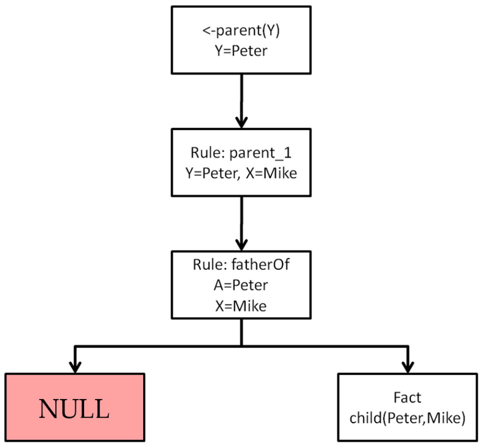


Figure 5.1: *Partial Prooftree Example*

partial prooftree for the atom $parent(mike)$ consists of only the null justification for $parent(mike)$.

5.1.5 Computation of Partial Prooftrees

Based on the concepts from abductive reasoning [90, 8] and on the algorithm sketched in [32], partial prooftrees can be computed as described in procedure `BACK-CHAIN-P`.

The algorithm presented extends the concepts of a backward chaining inference engine. The core change is that when a goal cannot be proven, the inference engine may assume it to be true in order to investigate whether it can find a likely proof with only a few assumptions. The decision on whether to carry on with the inference process is then taken based on the relative size of the goals that have already been proven, the goals that still need to be proven and the set of assumed goals.

As an input the algorithm takes a logic program, the list of atoms to be proven, a substitution and two sets of already processed goals and assumed goals, respectively. At the beginning of the inference process, the list of atoms consists of the atoms from the query in any order. The substitution is empty, also the sets of assumed goals and already processed goals contain no elements.

On being called the procedure first checks whether the inference process has been successful, whether $qlist$ is empty and whether hence all goals have been proven. If this is the case, the current substitution and the sets A_p and A_a are returned. These sets are not directly necessary as result, but

```
Data: The logic program $P$, the list of atoms to be proven $qlist$, a
      substitution $\theta$, a set of already processed goals $A_p$ and a set of
      assumed goals $A_a$
Result: A set of tuples consisting of a subsitution $\theta$, a set of processed
        goals and a set of assumed goals, i.e. $(\theta, A_p, A_a)$
if $qlist$ is empty then
    return $\{(\theta, A_p, A_a)\}$
else
    $q \leftarrow$ FIRST$(qlist)$
    $goalFailed = true$
    foreach $q'_i$ in P such that $(\theta_i) \leftarrow$ UNIFY$(q, q'_i)$ succeeds do
        answers $\leftarrow (\theta\theta_i, A_p \cup q'_i, A_a) \cup$ answers
        $goalFailed = false$
    end
    foreach rule $(q'_1 \leftarrow (p_1, ..., p_n))$ in P such that $(\theta_i,) \leftarrow$ UNIFY$(q, q'_i)$ do
        answers $\leftarrow$
        BACK-CHAIN-P$(P, [p_1\theta_i, ..., p_n\theta_i], \theta\theta_i, A_p, A_a) \cup$ answers
        $goalFailed = false$
    end
    if $goalFailed = true$ then
        $A_a = A_a \cup \{q'_1\}$
        if $f_s(qlist, A_p, A_a) < threshold$ then
            answers $\leftarrow (\theta\theta_i, A_p, A_a \cup q'_i) \cup$ answers
        end
    end
end
foreach $(\theta)$ in answers do
    toReturn $\leftarrow$ BACK-CHAIN-P$(P, $REST$(qlist), \theta, A_a, A_p) \cup toReturn$
end
return toReturn
```

Procedure BACK-CHAIN-P (P,$qlist$,θ,A_p,A_a)

would allow to later rank the results based on the number of assumed goals or the proportion of assumed goals to succesfully proven ones.

If $qlist$ is not empty, the first goal from the list is taken and processed. First the procedure attempts to directly unify the goal with the facts in the logic program. For every fact where this succeeds, the substitution and the sets of assumed and already proven goals are added to the set of answers.

In the next step the goal is matched against the heads of the rules in the logic program. For every rule for which it is possible to unify the head with the current goal, a recursive call to the BACK-CHAIN-PARTIAL procedure is made. This call takes as arguments the body goals from the rule with the substitution applied, the current substitution and the sets A_a and A_p. The results returned by this recursive call are added to the sets of answers.

If both the unification with facts and the unification with rule heads did not succeed at least once, the current goal is added to the list of assumed goals. A check is then done whether the proportion of assumed goals to other goals has not gotten too large; this is done to ensure that the inference engines tries only promising paths of reasoning. If the proportion is still acceptable[1], the current substitution and the sets A_a and A_p are added to the set of answers.

In the final loop of the procedure, a recursive call is done for every substitution in the set of answers. The purpose of these calls is to prove the remaining goals in $qlist$. The results from the calls are aggregated in a set that is then returned from the procedure.

Note that this kind of computation is not guaranteed to find every possible partial prooftree - not even all partial prooftrees that would satisfy the threshold. This limitation is a consequence of the algorithm trying to prune the search space as early as possible; it is caused by two factors:

- The abortion of some paths of reasoning based on a too high proportion of assumed goals to other goals (already proven and yet to be proven). This may lead to abandonment of reasoning paths that could later add many more goals and produce partial prooftrees below the threshold.
- The assumption of goals only after all attempts to prove a goal failed. This may cause some partial prooftrees to be omitted in cases where a goal succeeds but only with a substitution that later on causes a proof to fail.

[1]The cutoff used in the implementation was: (number of proven goals + number of goals to prove +10) divided by (number of assumed goals *10) must be larger or equal to 1. The best cutoff, however, is an empirical question that also depends on the nature of the rule base and one that implies a tradeoff between finding more partial prooftrees and runtime.

The tradeoff between the loss of some partial prooftrees caused by the pruning of the search space and the runtime of the inference engine is an empirical question that to a great extent depends on the structure of the rule bases; no final answer to this tradeoff was found in this thesis. Another possible course of action would be to replace the depth-first search strategy of this algorithm with a breath or even best-first strategy: first exploring the reasoning path that looks most promising, but returning to the less promising ones should the first fail. This very promising direction of research is, however, beyond the scope of this thesis.

5.1.6 Mutated Prooftree

In some cases there are proofs that would succeed after only a small syntactic change to the rule base; changes smaller than adding or removing entire atoms as stipulated by the partial prooftree. Consider the following example and the (partial) prooftree for the atom $parent(peter)$.

$parent(Y) \leftarrow fatherOf(Y, X)$
$parent(Y) \leftarrow fatherOf(Y, X)$
$fatherOf(A, X) \leftarrow male(A), child(A, X)$ # Rule fatherOf
$motherOf(A, X) \leftarrow female(A), child(A, X)$ #Rule motherOf
$mele(Peter)$ # note the typo - supposed to be 'male'
$child(Peter, Mike)$

In this case there are two equally valid partial prooftrees - a proof could be created by adding either $male(Peter)$ or $female(Peter)$ to the rule base. However, at the same time just changing one character in $mele(Peter)$ would also make a proof possible and would allow for the creation of a prooftree. The mutated prooftree introduced in this section is a concept to capture this notion of proofs that would succeed with very small syntactic changes to the rule base. In this example a small syntactic change is defined based on the edit distance between terms and predicate symbols. With such a notion the mutated prooftree is only able to identify faults caused by typos. However, the mutated prooftree can use arbitrary notions of similarity and nearness, for example based on a type hierarchy/ontology.

The mutated prooftree is defined analog to the normal prooftree at the beginning of this section, the only difference being that it uses the Herbrand model after a mutation ρ has been used to transform the rule base. Mutated prooftrees are scored by $f_{sim}(\rho)$, where only those with the highest scores should be presented to the user.

Definition 27 (Mutated Prooftree) *A mutated prooftree $G_m(A_x, P, \rho)$ exists for all atoms A_x and mutations ρ where A_x is in the Herbrand model of the rule base obtained by applying ρ to all rules in the rule base, i.e. $G_m(A_x, P, \rho) \equiv G(A_x, P\rho)$.*

Intuitively, a mutated prooftree captures the notion of a proof that would succeed after only small syntactic changes to the rule base. For practical applications mutated prooftrees can again be computed only heuristically - one technique of doing that is described in the next section.

Returning to the example from the begining of the section, multiple mutated prooftrees exist for the given rule base and the atom $parent(peter)$. One mutated prooftree is the expected $G_{m1}(parent(peter), P, \{mele/male\})$ and another one would be $G_{m2}(parent(peter), P, \{mele/parent\})$. The scoring function f_{sim} is used to establish the most likely mutated prooftree(s). In this example a scoring function based on the edit distance would correctly identify G_{m1} as the most likely prooftree and the developer can then be guided to check whether $mele(Peter)$ really is a fault or is actually intended.

5.1.7 Computation of Mutated Prooftrees

The algorithm to compute the Mutated Prooftree is also a variant of backward chaining, however, it uses a different unification routine that does not fail on not-equal predicates, function symbols or constants but rather proposes a mutation that would make the unification possible. Again a threshold is introduced that aborts a particular reasoning path in cases where the mutation similarity gets too low.

To describe the notion of the mutated prooftree, first the unification with mutation needs to be introduced. In order to do that effectively, the notions of subexpression and disagreement pair need to be defined first.

Definition 28 (Subexpression) *For every j the subexpression $s(a_x, j)$ of the atom a_x is the j-th subexpression of a_x. $s(a_x, j)$ is undefined, if j is larger than the number of subexpressions.*

For example for the atom $a_x = pred(f(g(a)))$, $s(a_x, 0)$ is $f(g(a))$ and $s(a_x, 2)$ is a. Based on this notion of subexpressions, disagreement pairs are defined.

Definition 29 (Disagreement Pair) *The disagreement pair $d(a_x, a_y)$ of two non-equal atoms a_x and a_y is the pair of subexpressions $s(a_x, j)$ and $s(a_y, j)$, where j is the smallest index such that $s(a_x, j) \neq s(a_y, j)$. A disagreement pair $(s(a_x, j), \emptyset)$ or $(\emptyset, s(a_y, j)$ is defined for the cases where $s(a_y, j)$ or $s(a_x, j)$ is not defined (because one atom has fewer subexpressions than the other).*

Intuitively, the disagreement pair for two atoms reflects the first non-equal subexpressions in these atoms. For example the disagreement pair for the atoms $pred(f(g(a)), c(a))$ and $pred(f(h(a)), c(f))$ is $(g(a), h(a))$. Note that there is always only one disagreement pair for each pair of atoms.

With these notions the adaptation of the unification algorithm can be presented.

```
Data: A pair of atoms $(a_x, a_y)$ and a mutation $\rho$
Result: A substitution $\theta$, and the possibly changed mutation $\rho$
Set $\theta$ to $\emptyset$
while There is a disagreement pair $d(a_x\rho\theta, a_y\rho\theta) = (s_x, s_y)$ do
    if Neither $s_x$ nor $s_y$ is a variable then
        $\rho = \{(s_x, s_y)\} \cup \rho$
    else
        Let $s_x$ be the variable in $(s_x, s_y)$
        if $s_x$ occurs in $s_y$ then return failed
        else
            $\theta = \{(s_x, s_y)\} \cup \theta$
        end
    end
end
return $\theta$,$\rho$
```

Procedure `UNIFY-M`((a_x, a_y),ρ)

The adapted unification algorithm continues to process the disagreement pair in $(a_x\rho\theta, a_y\rho\theta)$ while adding elements to ρ and θ until either the unification fails or there is no disagreement pair left. The unification fails only on the occur check - when a variable that is used for unification is already contained in the term it is supposed to be bound to; something not frequently encountered in practice[2] [144]. The difference between this unification procedure and normal unification is in the behavior on disagreement pairs that do not include a variable. In this case the two subexpressions (terms) are added to the mutation ρ while a normal unification would sim-

[2]In fact the optimized unification used in Prolog usually omits even this check in the interest of speed.

ply fail.

Note that while this procedure is fast[3], it is not guaranteed to find the smallest mutation that would unify two atoms. For example the unification of the two atoms $pred(a(b(c)), X)$ with $pred(a(d(c)), f(a))$ would result in $\theta = \{(X, f(a))\}$ and $\rho = \{(a(b(c)), a(d(c)))\}$ instead of the optimal $\rho = \{b, d)\}$. More complete unification algorithms with mutation are easily conceivable, but would also be considerably slower. As a final note it has to be remarked that while this unification is as fast as normal unification in the worst case, it is slower in practice. This is due to many of the indexing techniques normally used to prevent unnecessary calls to the unification function not working once mutation is included.

With the extended unification algorithm introduced, backward chaining with mutation can be defined. As input the procedure takes a logic program, the list of atoms to be proven, a substitution, and a mutation. At the beginning of the inference process, the list of atoms consists of the atoms from the query, the substitution is empty and the mutation is empty.

The procedure first checks whether the list of goals is empty; the inference process has been completed. If this is the case, the current substitution and the current mutation are returned as result.

In the cases where $qlist$ is not empty, the first goal from the list is taken and then processed. As a first step it is attempted to unify this goal with a fact in the program. If this succeeds and the mutation similarity is above a predefined threshold, the substitution and the mutation are added to the set of answers.

Next the unification of the atom with the heads of the rules in the program are attempted. For all rules where this succeeds and the necessary mutation has a similarity above the threshold, a recursive call is made to prove the unified rule's subgoals.

In the final loop of the procedure, a recursive call is done for every substitution in the answers. The purpose of these calls is to prove the remaining goals in $qlist$. The results from the calls are aggregated in a set that is then returned from the procedure.

Note that this kind of computation is not guaranteed to find every possible mutated prooftree - not even all that would satisfy the threshold. This is a consequence of the details of the unification procedure whose limits were described earlier.

The computation of mutated prooftrees and partial prooftrees can be com-

[3]Like normal unification: quadratic to the size of the atoms being unified, or linear (but incorrect) if the occur check is omitted [144].

```
Data: The logic program $P$, the list of atoms to be proven $qlist$, a
      substitution $\theta$ and a mutation $\rho$
Result: A set of tuples, each consisting of a subsitution $\theta$ and a mutation $\rho$,
        i.e. $(\theta, \rho)$
if $qlist$ is empty then
    return $\{(\theta, \rho)\}$
end
else
    $q \leftarrow$ FIRST$(qlist)$
    foreach $q'_i$ in P such that $(\theta_i, \rho_i) \leftarrow$ UNIFY-M$(q, q'_i, \rho)$ succeeds do
        if $f_{sim}(\rho_i) > threshold$ then
            answers $\leftarrow (\theta\theta_i, \rho_i) \cup$ answers
        end
    end
    foreach rule $(q'_1 \leftarrow (p_1, ..., p_n))$ in P such that
    $(\theta_i, \rho_i) \leftarrow$ UNIFY-M$(q, q'_i, \rho)$ do
        if $f_{sim}(\rho_i) > threshold$ then
            answers $\leftarrow$
            BACK-CHAIN-MUT$(P, [p_1\theta_i, ..., p_n\theta_i], \theta\theta_i, \rho_i) \cup$ answers
        end
    end
end
foreach $(\theta, \rho)$ in answers do
    toReturn $\leftarrow$ BACK-CHAIN-MUT$(P,$ REST$(qlist), \theta, \rho) \cup toReturn$
end
```

Procedure BACK-CHAIN-MUT (*P,qlist,θ,ρ*) bined into one algorithm. An atom is then assumed only after it could not be proven; no atom is assumed if a small mutation is sufficient to proof a goal.

5.2 Implementation

The previous section introduced rules, rule firings, the static structure and the dynamic structure of the rule base as components for explorative debugging. This section now presents the implementation of the Explorative Debugging concepts in two programs:

- As part of the **Trie!** (Transparent RDF Inference Engine) application. This debugging prototype supports rules on top of RDF (as defined in section 2.3.4) and implements all variants of the static and dynamic structure introduced in the previous section. This debugger was used as the basis for the evaluation presented later in section 5.3.
- As the **Inference Explorer** for F-logic, integrated into the Dark Matter Studio system. This debugger does not support partial and mutation extended prooftrees, but is more mature and supports some additional features, such as the explanation of built-ins.

This section will start with the description of the newer Trie! system; a short description of the additional features of the Inference Explorer will be given after that.

5.2.1 The Trie! Framework

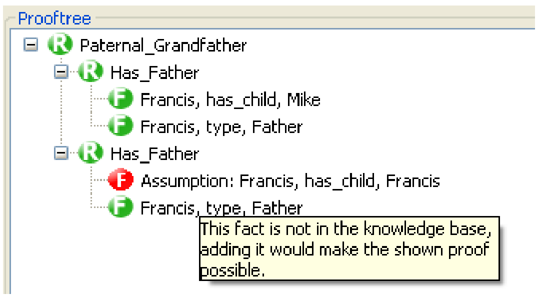
The concept of Explorative Debugging is implemented in the Trie! (**T**ransparent **R**DF **I**nference **E**ngine[4]) program presented here. The Trie! program is open source and available at `http://code.google.com/p/trie-rules/`. It is an Eclipse RCP (Rich Client Platform) [60] application consisting of an Explorative Debugger, a procedural debugger, a custom inference engine, a simple visualization of the (mutation extended) rule base dependency graph and tools to load and change RDF and rule definition files.

The **procedural debugger** is included to support the comparative evaluation between the Explorative and procedural debugging paradigm. The **dependency graph visualization** uses the Zest [54] toolkit to show the depends-on graph. Finally **file management** tools are included for loading RDF and rule files into the knowledge base. An integrated text editor allows to change these files. The program is able to read RDF/XML [10] and Turtle [11] files. Rule files are expected in the rule format used in the Jena Semantic Web toolkit [30], with the additional option for rule comments. Inference engine and Explorative Debugger are described in more detail in the sections below.

[4]Named in recognition to the pioneering work of the Transparent Prolog Machine [56].

The Trie! Inference Engine

The custom Trie! inference engine is built specifically to facilitate the easy creation of debugging applications (e.g. by sacrificing memory efficiency to have an easily accessible internal state). The inference engine currently supports the rule language described in section 2.3.4; it currently has no support for built-ins. The evaluation strategy used is backward chaining without tabling[5].

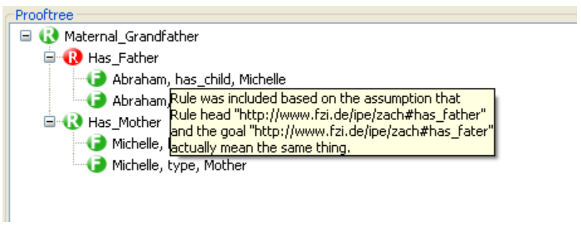


```
Prooftree
⊟ Ⓡ Paternal_Grandfather
   ⊟ Ⓡ Has_Father
      Ⓕ Francis, has_child, Mike
      Ⓕ Francis, type, Father
   ⊟ Ⓡ Has_Father
      Ⓕ Assumption: Francis, has_child, Francis
      Ⓕ Francis, type, Father
```

This fact is not in the knowledge base, adding it would make the shown proof possible.

Figure 5.2: *A Partial Prooftree in the user interface*

The Trie! inference engine realizes the algorithms described in section 5.1.5 in the creation of partial prooftrees: on failure to prove a goal the inference engine does not backtrack, rather it continues to look for partial proofs that would succeed if this goal would be true. The proportion of goals that are already successfully proven, the rules used, and the goals still on the stack to the goals assumed to be true are used as a heuristic to guide the inference engine; once the proportion of assumed goals becomes too high the inference engine backtracks. Further sorting and filtering is done after a query has been processed; e.g. prooftrees are sorted by the proportion of facts and rules to assumptions; no partial prooftree is returned for a result for which there is also a complete prooftree. The user interface used to display partial prooftrees is shown in figure 5.2.



```
Prooftree
⊟ Ⓡ Maternal_Grandfather
   ⊟ Ⓡ Has_Father
      Ⓕ Abraham, has_child, Michelle
      Ⓕ Abraham,
   ⊟ Ⓡ Has_Mother
      Ⓕ Michelle,
      Ⓕ Michelle, type, Mother
```

Rule was included based on the assumption that Rule head "http://www.fzi.de/ipe/zach#has_father" and the goal "http://www.fzi.de/ipe/zach#has_fater" actually mean the same thing.

Figure 5.3: *A mutation extended prooftree in the user interface*

Mutated prooftrees are generated through a modification of the unification algorithm (detailed in section 5.1.6) at the core of the Trie! inference engine. Instead of failing on not-equal ground terms the unification algorithm returns the number of edit operations (Levenshtein distance) that would need to be performed on ground terms in order to unify two atoms.

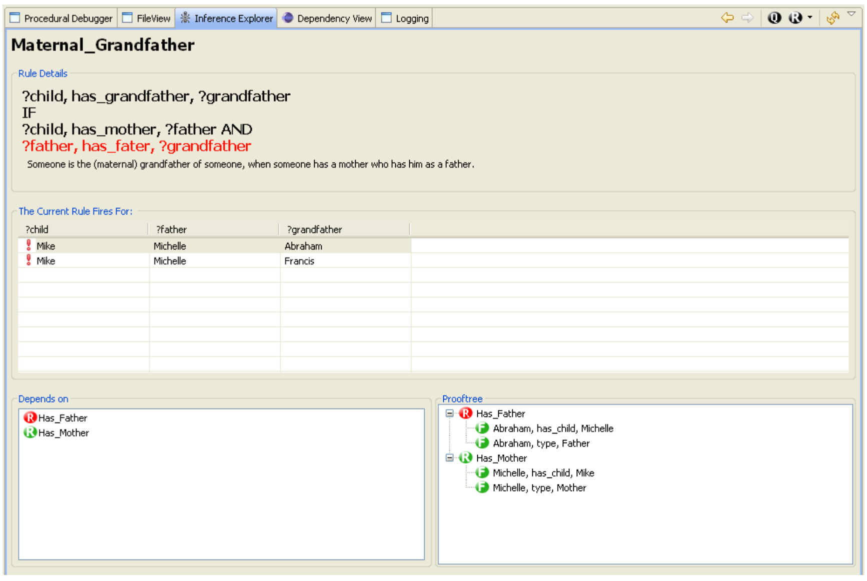
[5]Meaning in particular that the inference engine is not able to deal with rule cycles.

Because of the simple structure of the rule base[6] this change does not complicate or slow down the unification algorithm excessively. The inference engine backtracks only when the number of edit operations exceeds a specified threshold (see section 5.1.5 and 5.1.7). The same technique is used to compute the extended depends-on connections between rules. The user interface used to display mutated prooftrees is shown in figure 5.3.

Partial and mutated prooftrees are created at the same time and may in fact be combined, i.e. the inference engine can return a proof that assumes one goal and is also based on the almost-unification of two atoms (i.e. made possible by a small number of edit operations).

The Trie! Explorative Debugger

The user interface of the explorative debugger in the Trie! application consists of five main areas: the navigation controls, rule details, rule firings, depends-on connections and prooftree. A screenshot of the user interface is shown in figure 5.4.



Maternal_Grandfather

Rule Details

?child, has_grandfather, ?grandfather
IF
?child, has_mother, ?father AND
?father, has_fater, ?grandfather

Someone is the (maternal) grandfather of someone, when someone has a mother who has him as a father.

The Current Rule Fires For:

?child	?father	?grandfather
! Mike	Michelle	Abraham
! Mike	Michelle	Francis

Depends on

- Has_Father
- Has_Mother

Prooftree

- Has_Father
  - Abraham, has_child, Michelle
  - Abraham, type, Father
- Has_Mother
  - Michelle, has_child, Mike
  - Michelle, type, Mother

Figure 5.4: *User interface of the Trie! Explorative Debugger*

The symbols at the top right of the interface are the **navigation controls** that allow to jump to any rule, jump to the query or to reload data from files. A configuration menu enables to show or hide the namespace part of Uri's.

[6]I.e. it does not include function symbols and only the ternary predicate symbol *triple*.

The **rule details view** at the center of the interface shows the current rule and its comment. The body atoms of the rule are colored depending on whether an atom alone is satisfiable; e.g. in the screenshot the atom ?father, has_fater, ?grandfather (note the typo in has_fater) is colored red because it is unsatisfiable. Clicking on an atom in the rule details view temporarily disables a rule atom, the debugger will then show how the rule would function without this atom.

Below the rule details the **rule firings** are shown. A red exclamation mark is used to mark results for which there is only a partial and/or mutated prooftree. The variable bindings selected in this part determine the prooftree that is shown.

In the bottom left of the interface the current **depends-on** connections are shown. The rules shown here are based on the extended depends-on graph. Rules are shown in red, when the connection to them is only afforded by a (non-empty) mutation. A double click on any rule shown, opens this rule in the Explorative Debugger.

At the bottom right the prooftree for the currently selected result is shown. For partial and mutated prooftrees, the mutated or assumed parts are shown in red and tooltips give more information about them (see also figures 5.2 and 5.3).

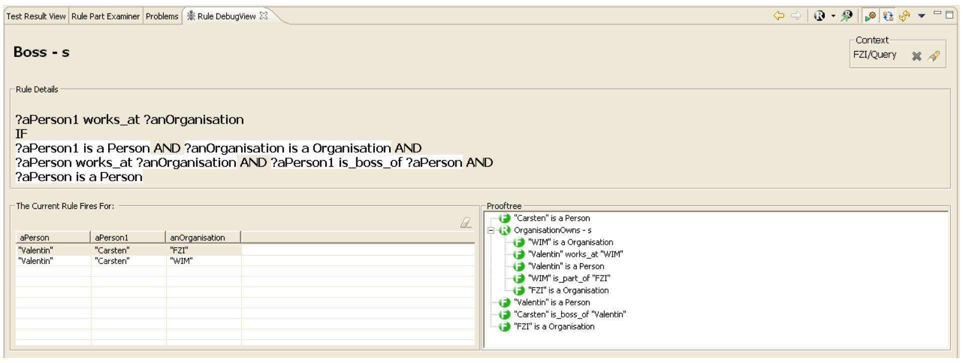
5.2.2 Inference Explorer

The Inference Explorer is the Explorative Debugger that was implemented as part of the Dark Matter Studio system in Project Halo. The Inference Explorer supports the debugging of F-logic rule bases and uses the Ontobroker [50, 3] as inference engine. Partial and mutated prooftrees are not supported by Ontobroker and hence cannot be shown in this system.

The debugger is implemented as a plugin for Eclipse that works within DMS. For the user interface it relies exclusively on SWT [161]. It is implemented following the well known model-view-controller [95] pattern. The different parts of the debugger (details view, information view etc.) are implemented as independent views that do not know of each other, but that share one model and controller. The model is described in more detail later in this section.

A screenshot of the Inference Explorer is shown in figure 5.5. The basic user interface is similar to that of the Trie! debugger described earlier (see figure 5.5); although it consists only of three main elements:

- The display of rules and rule parts at the top. Again this part allows



Boss - s

Rule Details

```
?aPerson1 works_at ?anOrganisation
IF
?aPerson1 is a Person AND ?anOrganisation is a Organisation AND
?aPerson works_at ?anOrganisation AND ?aPerson1 is_boss_of ?aPerson AND
?aPerson is a Person
```

The Current Rule Fires For:

aPerson	aPerson1	anOrganisation
"Valentin"	"Carsten"	"FZI"
"Valentin"	"Carsten"	"WIM"

Prooftree

```
"Carsten" is a Person
OrganisationOwns - s
    "WIM" is a Organisation
    "Valentin" works_at "WIM"
    "Valentin" is a Person
    "WIM" is_part_of "FZI"
    "FZI" is a Organisation
"Valentin" is a Person
"Carsten" is_boss_of "Valentin"
"FZI" is a Organisation
```

Figure 5.5: *User Interface of the Inference Explorer*

to deactivate rule parts and rule parts are colored based on their satisfiability.

- At the bottom left rule firings are shown.
- At the bottom right one pane displays information depending on what is selected. It shows the depends-on connections when nothing or a rule atom is selected. It shows the prooftree when one of the rule firings is selected and it shows documentation text in case the selected rule atom is a built-in.

The Inference Explorer does not support mutation extended prooftrees and partial prooftrees, however, it does have some features not included in the less mature Trie! Explorative Debugger, these are:

- A 'Debugging Context'
- A simplified prooftree display that hides F-logic axioms
- A configurable rendering of rules and facts
- An integrated documentation for built-ins used in the rules.

Each of these features will be shortly described in the following paragraphs.

Debugging Context

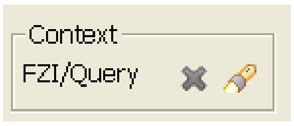


Figure 5.6: *The debugging context information and toolbar*

For the debugger it is relevant whether it should be executed within a test set. A rule normally not firing at all, may fire when the facts from a test set are considered; also the query from a test case is not normally in the rule base and could not be debugged. For these reasons the debugger knows about the test case from which it was called; this is called the *context* for the debugger. The context contains the test case query and all setup facts from the test set. The context of a debugger is clearly shown to the user in a box in the top right of the debugger (see figure 5.6 and figure 5.5). Inside the box the system displays the name of the current test set and test query; FZI and Query in this example. The debugging context also offers two buttons with commands: the cross to the left "ends the context". Ending the context means that the debugger henceforth stops considering the setup facts from the test and will not show the query anymore. The flashlight to the right is a quick way to navigate to the test case query - clicking it instantly opens the query in the debugger.

Simplified Prooftree Display

The Ontobroker inference engine evaluates F-logic after the translation into normal logic - as described in the section 2.2. Doing this introduces a mismatch between the prooftree returned by the inference engine and the user level understanding of F-logic. For example the prooftree usually contains the axiom rules that the user should not need to know about or it may contain multiple rules that have been created from the one rule the user specified. This is further complicated by the inclusion of numerous implementation specific operations in the prooftree, e.g. elements representing the access to a database. To deal with this mismatch between the user level entities and entities of the inference engine, the debugger transforms the prooftree before it is shown to the user. In this transformation

1. All implementation specific nodes are removed;

2. Some axioms, e.g. subclassTransitivity1 (see section 2.2), are removed;

3. Prooftree nodes representing instances of multiple rules created from one user rule are joined again;

4. Prooftree nodes representing the application of axioms not removed by step 2, are replaced by natural language explanations.

When prooftree node A, with parent node B and children $C_1, ..., C_n$ is removed from the prooftree, all children $C_1, ..., C_n$ are then reconnected to the parent node B. When prooftree nodes are joined together, again the children of all joined nodes become children of joined node. The new node is also connected to all parents of the nodes that were joined[7].

Through this transformation a prooftree is created that faithfully represents the inference process without requiring the user understand or even know about the transformation going on in the background.

Configurable Rendering

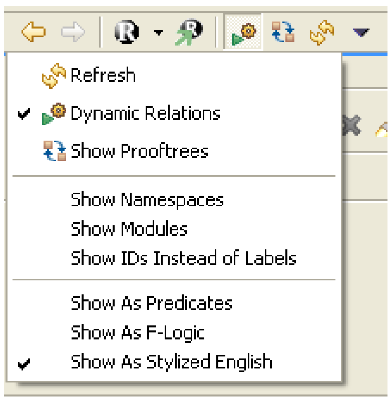


Figure 5.7: *The configuration menu of the Inference Explorer*

Dark Matter Studio includes a lexical layer that allows to show the facts and rules while hiding the internal ids, namespaces and modules. The debugger supports this lexical layer. However, because not all faults are visible at this level, it also includes the option to change its display to also include some or all of the normally hidden information. All these options affect all parts of the debugger where such entities are displayed; i.e. they simultaneously change the rule name, rule details, results, depends on and the prooftree. The meaning of the options in detail:

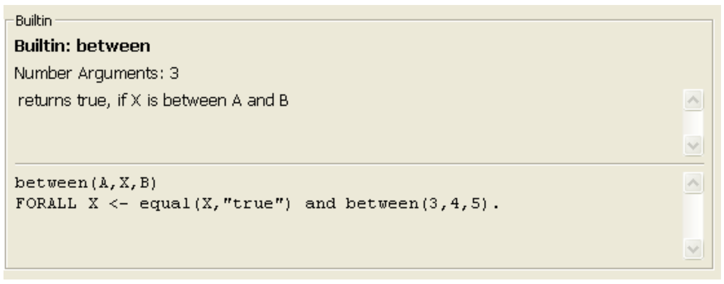
- **Show Namespaces:** Shows or hides the namespaces in the id's of entities.
- **Show Modules:** Shows or hides the modules of statements and rules.
- **Shows IDs instead of Labels:** DMS uses automatically generated ids for most entities in the knowledge base and has labels defined for the ids. For expert users, however, it is sometimes needed to look at the actual ids.

[7]Note that this means that the prooftree may stop being a tree, may become a directed graph. The nature of the nodes that get joined, however, means that this is normally not the case; it would also not break the program.

- **Show As Predicates:** Shows everything translated into a normal logic program.
- **Show As F-logic:** Shows everything in standard F-logic syntax.
- **Show As Stylized English:** Shows all entities from the knowledge base in a simple form of stylized english. This representation of F-logic as stylized english is used throughout DMS and hence familiar to the user.

The defaults for all of these options is the simplest representation and it is assumed that virtually all domain experts will exclusively use the default configuration. To also support expert users and to aid in the development of the DMS system, however, it was needed to allow the look at lower abstraction level.

Documentation of built-ins



Builtin: between

Number Arguments: 3

returns true, if X is between A and B

```
between(A,X,B)
FORALL X <- equal(X,"true") and between(3,4,5).
```

Figure 5.8: *Built-in information in the Inference Explorer*

A particular challenge for debugging are those parts of the rule base that access built-ins - parts of the rule base realized as separate programs; usually created in a different programming language. To support the developer in using these and debugging faults appearing in the user of built-ins, the Inference Explorer can display information about a selected built-in (see figure 5.8).

The challenge here is that the built-ins are changing relatively frequently, that these documentation could be used in different parts of DMS and that the person writing the documentation is often not a programmer. For these reasons it was decided to describe the documentation of built-ins in simple XML files that could be easily edited and to include interfaces to allow access to this documentation from all parts of Dark Matter Studio.

The following example shows a very short built-in documentation file containing only the description for the *between* builtin:

```
<builtinSet>

  <builtin>
    <description>
     returns true, if X is between A and B
    </description>
    <syntax>between(A,X,B)</syntax>
    <example>
     FORALL X :- equal(X,"true") and between(3,4,5).
    </example>
    <name>between</name>
    <numberArgs>3</numberArgs>
  </builtin>

</builtinSet>
```

BuiltinSet is the topmost container element containing all built-ins. The description for each built-in is enclosed in a built-in element. The description, syntax and example elements are directly shown to the user. The last two, name and the number of arguments are used to identify a predicate as built-in.

5.2.3 Debugging Walkthrough

This section describes a simply fictive debugging session with the Inference Explorer and the testing interface described in the previous chapter. It starts by describing the contents of the knowledge base that is to be debugged. It then shows the definition of a test case and how the debugger can be used to identify the problem. This section assumes that the reader is familiar with the graphical notation used for the definition of test cases (described in section 4).

Contents of the Knowledge Base

The knowledge base for this walkthrough defines some simple relations between organizations, persons employed at organization and subordinate relationships. This toy knowledge base enables the (questionable) inferences that a person is employed at an organization because its superior is employed there and, through a second rule that a person can be said to work at an organization if it is known that he works at an entity that is part of the organization.

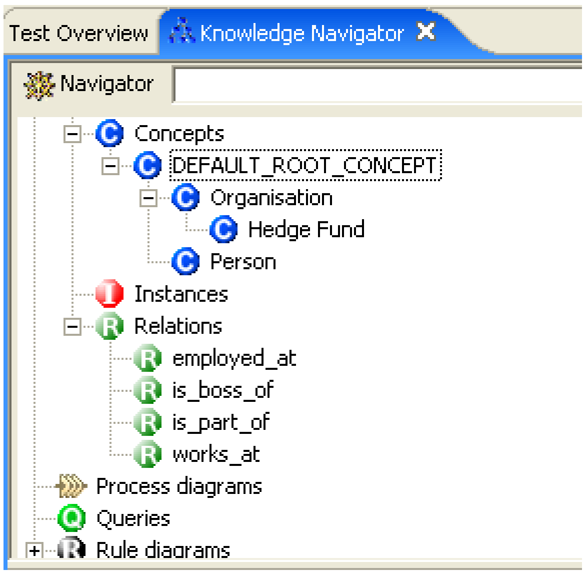


Figure 5.9: *The ontology for the debugging walkthrough*

The ontology for this example only defines three concepts: Organization, Hedge Fund and Person. Hedge Fund is defined as a subclass of organization. All in all there are four relations, the relations "is boss of" between persons, the relation "is part of" between organizations and "works at" and "employed at" from person to organization. A view of the ontology in the knowledge navigator is shown in Figure 5.9.

The knowledge base for the example consists of two rules: rule 'OrganisationOwns' and rule 'Boss', both are shown in figure 5.10. Rule OrganisationOwns defines that a person that works at one organization A that is owned by another organization B, is said to also work at organization B; unless organization B is a hedge fund. The second rule, rule Boss, states that if a person A has a boss B and he is employed and an organization C, then person A also has a works_at relation to organization C. The reader may note that the second rule, rule Boss, is faulty. Throughout the knowledge base the works_at relation is used to denote that a person works at an organization. Rule Boss, however, in one part uses the employed_at relation instead. This is the fault that will be debugged in this walkthrough.

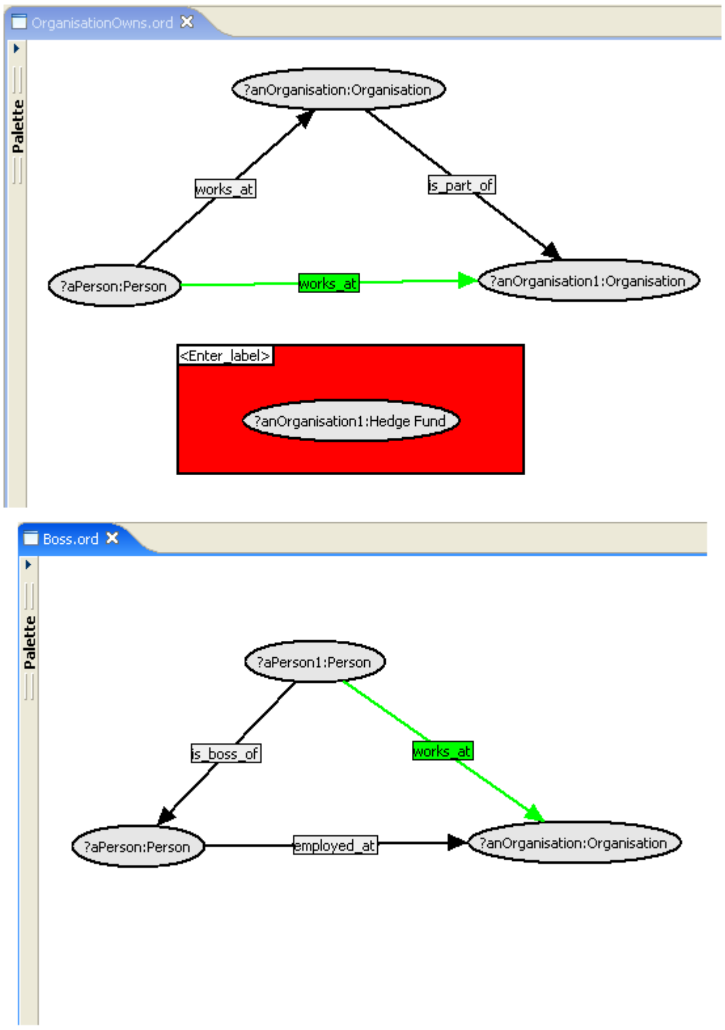


Figure 5.10: *The two rules for the debugging walkthrough*

Tests

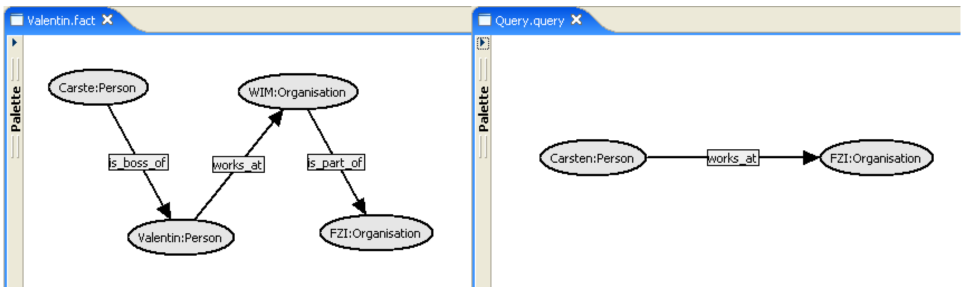


Figure 5.11: *Test fact and query definition for the debugging walkthrough*

A debugging session starts either directly from the definition of a rule or from a test case that does not return the expected result. The latter case is used for this walkthrough. Figure 5.11 shows the definitions of facts and query. The facts describe an organisation WIM that is part of another organisation FZI. A person Valentin has a works_at relation to WIM, it is stated that another person Carsten is the boss of Valentin. The query asks whether it is true that Carsten works at FZI. The assumption of the user is that it is indeed true, that the "OrganisationOwns" rule can infer a works at relation between Valentin and FZI and that the "Boss" rule can then infer that Carsten works at FZI because his subordinate does. However, the fault in the rules prevent this from working.

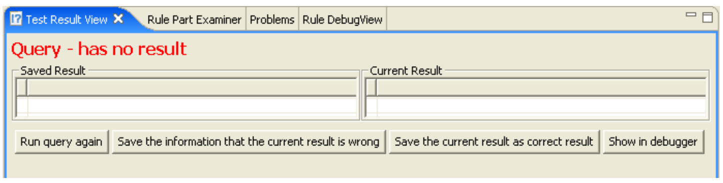
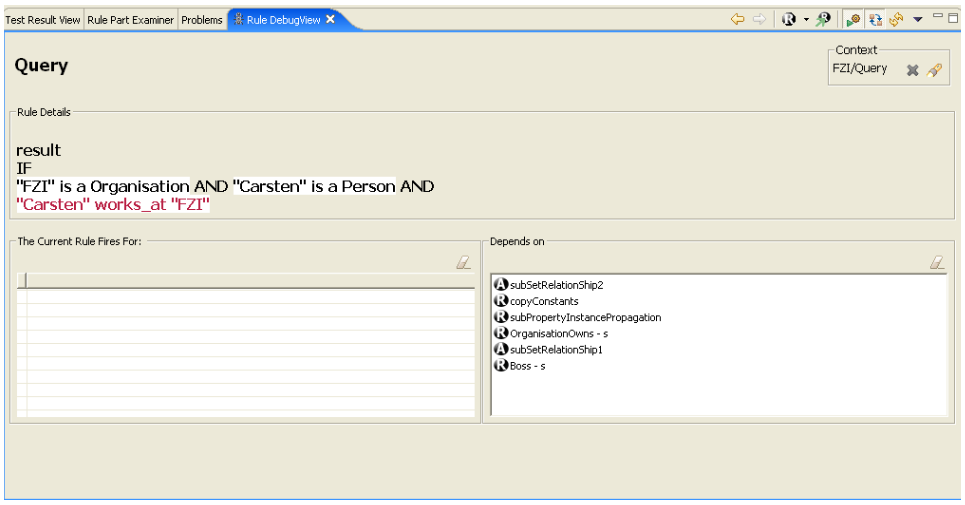


Figure 5.12: *Test result view for the debugging walkthrough*

After defining the test, the user runs it by pressing the "Run Query" button in the Test Result View. The test returns no result instead of the expected "true" (see Figure 5.12). The user presses the "Show in debugger" button in the lower right corner of the test result view to start the debugger.

Debugging


Test Result View | Rule Part Examiner | Problems | Rule DebugView

Query

Context
FZI/Query

Rule Details

result
IF
"FZI" is a Organisation AND "Carsten" is a Person AND
"Carsten" works_at "FZI"

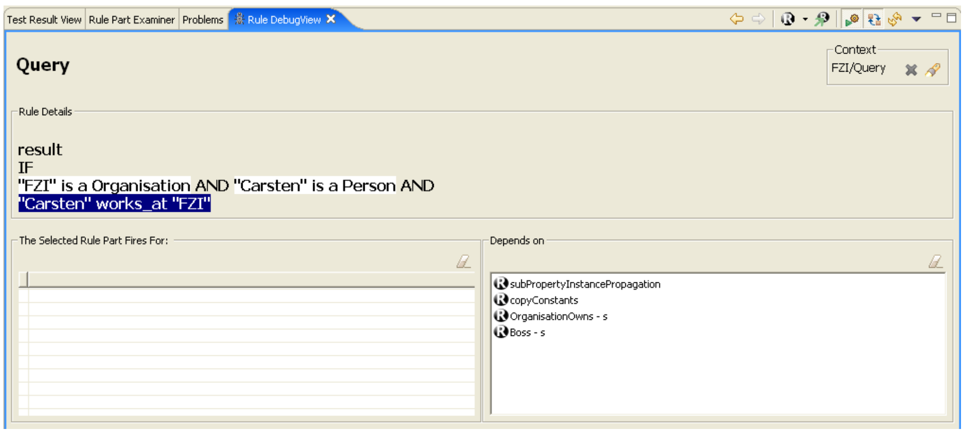
The Current Rule Fires For:

Depends on

subSetRelationShip2
copyConstants
subPropertyInstancePropagation
OrganisationOwns - s
subSetRelationShip1
Boss - s

Figure 5.13: *The debugger has identified the only unsatisfiable atom in the query*

Initially the debugger shows information about the failed test query (see Figure 5.13. It shows a stylized English representation of the query, the current results (none in this case) and the rules that could contribute to a result. One of the rule atoms is colored in red, indicating that it is the only atom that is not satisfiable when viewed alone. It is very likely that the problem lies with this atom. The user clicks on it.


Test Result View | Rule Part Examiner | Problems | Rule DebugView

Query

Context
FZI/Query

Rule Details

result
IF
"FZI" is a Organisation AND "Carsten" is a Person AND
"Carsten" works_at "FZI"

The Selected Rule Part Fires For:

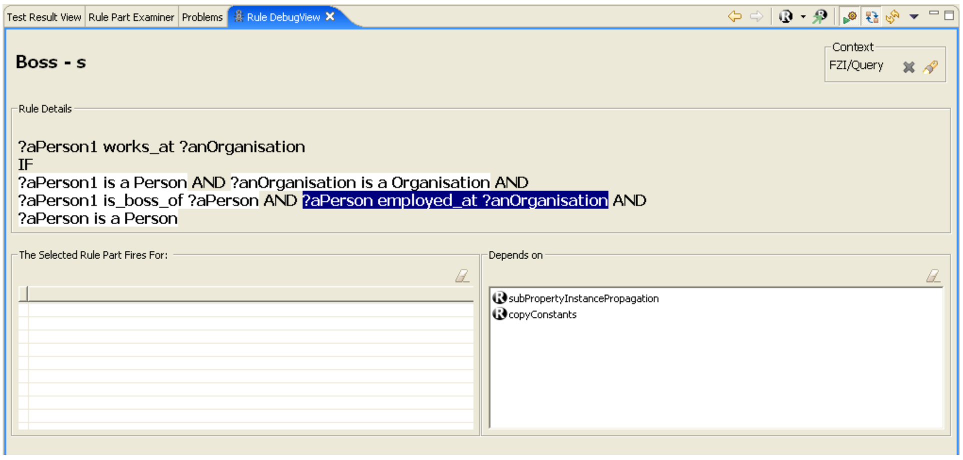
Depends on

subPropertyInstancePropagation
copyConstants
OrganisationOwns - s
Boss - s

Figure 5.14: *The user has selected the unsatisfiable atom in the query*

As a response to the selection of the rule atom the lower part of the debugger changes (see Figure 5.14). The left part now shows everything that satisfies this atom (as opposed to the whole rule body) - still nothing is shown since this is an unsatisfiable rule part. The lower right part of the debugger now shows only the other rules that this rule part could depend on. The user sees the "Boss" rule in the depends-on view, remembers that it should be involved in generating a result and double clicks on it.

Double clicking on the Boss rules takes the user to the debug view of this rule; the debugger now displays information about the Boss rule. The debugger highlights the one unsatisfiable atom in the rule Boss in red and the user selects this atom (see figure 5.15). Please note that this atom contains the fault that had been introduced in the knowledge base.



?aPerson1 works_at ?anOrganisation
IF
?aPerson1 is a Person AND ?anOrganisation is a Organisation AND
?aPerson1 is_boss_of ?aPerson AND ?aPerson employed_at ?anOrganisation AND
?aPerson is a Person

subPropertyInstancePropagation
copyConstants

Figure 5.15: *The user has selected the only unsatisfiable atom in the query*

Again the lower part of the debugger changes to show the details for this atom. There is no binding for this rule and only two unrelated rules are shown in the depends-on part. The user realizes that there must indeed be a problem with this atom, since it does not depend on the OrganisationOwns rule as she expected. The user presses on the "Open Rule" button and the system opens the Boss rule in the rule editor, the atom currently selected in the debugger is highlighted in the editor (see Figure 5.16).

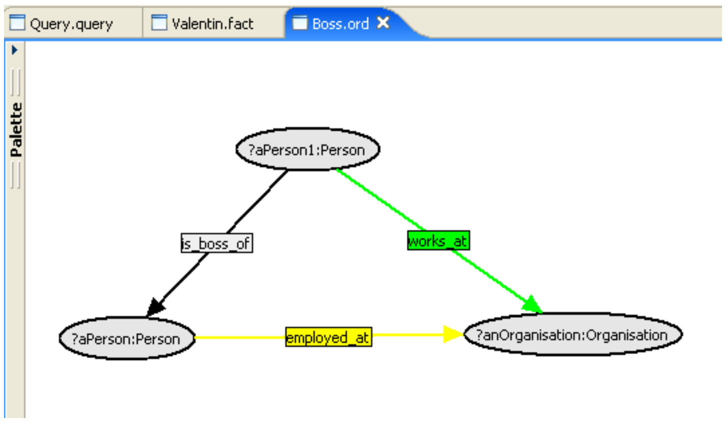


Figure 5.16: *The atom identified by the user is highlighted in yellow the rule editor*

The debugger has helped the user to find the exact position in the rule base where the error lies. It also supports the user in finding it in the editor where she can change it immediately.

5.3 Evaluation

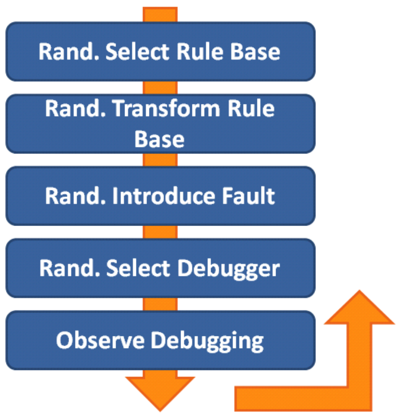


Figure 5.17: *Debugging evaluation design*

The hypothesis to evaluate is that Explorative Debugging improves a developer's effectiveness in finding faults in rule bases. An improved effectiveness could be either a reduced time needed to identify a bug or an improved accuracy in doing so. To evaluate this hypothesis it was decided to compare Explorative Debugging to procedural debugging - the most widely deployed debugging paradigm for rule bases. During the evaluation random faults would be introduced in a rule base and a comparison would be done of the effectiveness of developers in finding the fault based depending on what kind of debugger they used. In detail the setup for the evaluation is the following (see also figure 5.17):

1. A rule base is selected randomly.
2. The rule base is transformed randomly without impairing its function. This is needed since the same developer will debug the same rule base with different faults multiple times; this transformation ensures that he or she cannot simply memorize the correct rule base. The exact nature of this transformation is described later.
3. A random fault is introduced into the rule base; this is also described in more detail later.
4. A debugger is selected at random, in this case either a procedural (see section 5.4.1) or the Trie! Explorative Debugger.
5. The debugging process is observed and the fault identified by the user is saved. Later this is compared to the seeded fault.

All steps of this debugging setup are performed automatically. For this purpose one stand alone Java program has been created for the rule base transformation and mutation. Also an evaluation runner was created as another plugin for the Trie! program, its responsibility is the selection of rule base and debugger and the observation of the user's actions.

Two very small rule bases were used for this evaluation, each consisting of only 5 rules and a small number of facts. The rule bases are shown in the appendix B and C. Both rule bases are created with the RDF processing rule language described in section 2.3.4. For both rule bases a query is given as the start point for the debugging process.

This process is repeated and the results for the two kinds of debuggers are compared. Similar experimental designs have been used by the Software Engineering Research Center at Purdue University in the evaluation of a debugging oracle assistant [154] and by the NASA in the evaluation of debugging techniques for the Martian rover software [21], although the latter seeded not random faults but actual bugs that had been removed from earlier versions of the software.

5.3.1 Fault Seeding

Fault seeding is a four step process:

1. The random selection of a rule atom. Every rule atom, head or body had equal chance of being selected. In this evaluation the focus was on rule debugging in a narrow sense, hence no faults were seeded in facts. The query was included here, i.e. faults were also seeded in the body of the query.
2. The random selection of a term. In the RDF rule language used for the evaluation every atom has exactly three terms and each of these had the same chance of being selected.
3. A random change to this term. One out of six operations was chosen to randomly transform the term. The operations are detailed below.
4. A checking of the rule base. The rule base is written to a file, parsed and the results for the test query are computed. This checking process is described in more detail later.

The six operations for changing terms are the following:

- Changing a URI to a literal or vice versa.
- Changing a variable to a URI or vice versa.
- Removing a random character from the term.
- Adding a random character to the term.
- Replacing a term by a synonym, taken from a list that contains multiple synonyms for the URIs, variables and literals in the program; for example this operation could change an URI "..#Male" to "...#masculine", "...#Masculine" or "...#male".
- Selecting a second term from the same atom and swapping these.

In the final step the rule base is checked whether it is still syntactically correct and whether the query actually shows an error, whether the seeded fault changes the result. The results for this checking process for 1000 seeded faults is shown in figure 5.18. It shows that in almost half of the cases the query is not affected by the seeded fault. In a further 10% the fault causes a syntactic problem (such as a literal in predicate position) and in 5% it causes a rule loop (which the Trie! Explorative Debugger cannot process[8]). These almost 65% of the rule bases are discarded. In 75% of the remaining rule bases the test query has no results (no-result-case) and in 15% a result different from that of the rule base without the seeded fault.

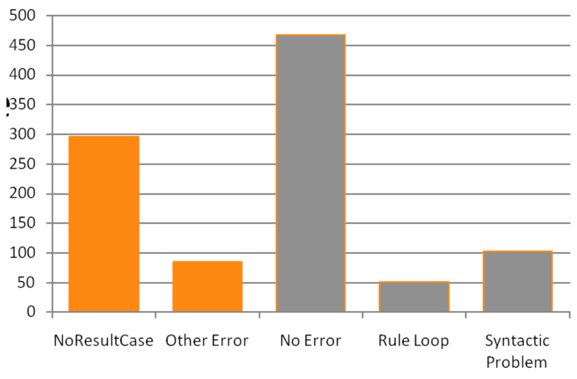


Figure 5.18: *Results from checking the rule base after fault seeding*

5.3.2 Rule Base Transformation

During the evaluation the same developers were asked to debug multiple different faults from the same rule base. In order to prevent them from finding these faults simply by comparing the rule base to their recollection of the correct one, the rule base is transformed before each iteration of the evaluation. In this transformation the majority of terms in the rule base is replaced with synonyms. In addition one typo is introduced. All changes done to the rule base in this step are applied in exactly the same way to all rules - hence the functionality of the rule base is not impaired by these changes.

Consider as an example the following rule:

```
[Rule_Pool_For_Kids:
  (?X, zach:suitedFor, zach:Kids)
<-
  (?X,rdf:type,zach:Apartment)
  (?X,zach:has_property,zach:Pool)
```

[8]Since it does not implement tabling.

```
]
```

The rule base transformation can turn this into the following rule[9].

```
[Rule_Pool_For_Kids:
  (?X,zach:suitedFor,zach:Children)
<-
  (?X,rdf:type,zach:Flat)
  (?X,zach:has_attribute,zach:Pool)
]
```

Here 'Kids' where replaced by 'Children', 'Apartment' by 'Flat' and 'has_property' by 'has_attribute' - making it generally impossible to find the fault in the rule simply by looking at isolated rules; in particular remember that a fault can be a replacement of a term by a synonym using the same list of synonyms used for the transformation. Also note that the transformation additionally introduces one random typo (adds or removes one character) that is made consistently across the whole rule base. After the transformation the rule bases were again checked for syntactic validity and whether the result truly was unchanged by the transformations - both conditions were met in 100% of the cases.

5.3.3 Evaluation Results

The results from repeating the described evaluation for 112 times show Explorative Debugging to be faster and more accurate in identifying faults.

With Explorative Debugging 91% of the faults were identified correctly; i.e. the user entered a free text description that correctly names the atom and term affected by the seeded fault. In 5% of the cases the user correctly identified the atom, but attributed the fault to a wrong term. In only 1.7% percent (one case) the user gave up and in another case he gave an incorrect answer. The overview of these results is shown in figure 5.19.

With procedural debugging still 74% of the faults were identified correctly; in one further case the fault was identified almost correctly. In 15% of the cases the user gave up. This high number of cases where the user gave up is due to the fact that with a procedural debugger alone it is almost impossible to debug cases that affect the unification of a body atom with another's rule head, since with a procedural debugging the failure to unify cannot be explored further. The user then has some idea in which part of the rule base the fault lies, but cannot identify it at the level of granularity expected for this evaluation. In 9% of the evaluation runs the fault was not identified correctly.

[9]This is an example from an actual file used in the evaluation.

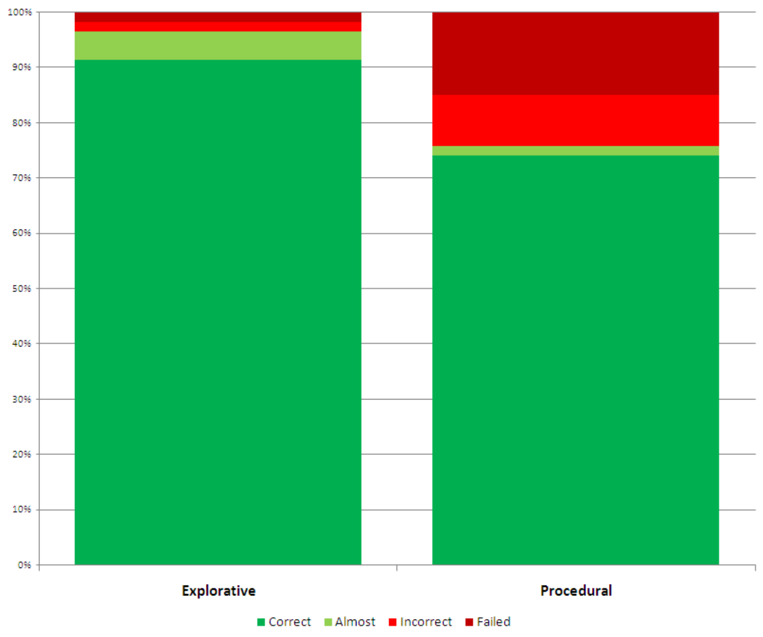


Figure 5.19: *Correctness in identifying faults using the procedural and Explorative debugger*

The average time for users to identify a fault (excluding those cases where the user gave up) is 41 seconds for the Explorative and 69 seconds, or a little longer than one minute for the procedural debugger - very short times explained by the small size of the rule base being debugged. The times for all evaluations runs are shown in figures 5.20 and 5.21, respectively. The difference between the average values is significant with a significance level of 0.05.

5.3.4 Evaluation Observations

In addition to the quantitative measurements described in the previous section, a number of qualitative observations were made in the evaluation; these concerned mutated and partial prooftrees in general, the difficulty of the no-result-case, attributing failures to single atoms and the relation between faults and similarity function in the mutated prooftrees. The following paragraphs will describe each of these in more detail.

During the evaluation **mutated and partial prooftrees** indeed proved to be often able to sum up the reason for the failure of a rule base to conclude an expected result in one picture; often identifying the faults was possible from just looking at the mutated or partial prooftree for the expected result,

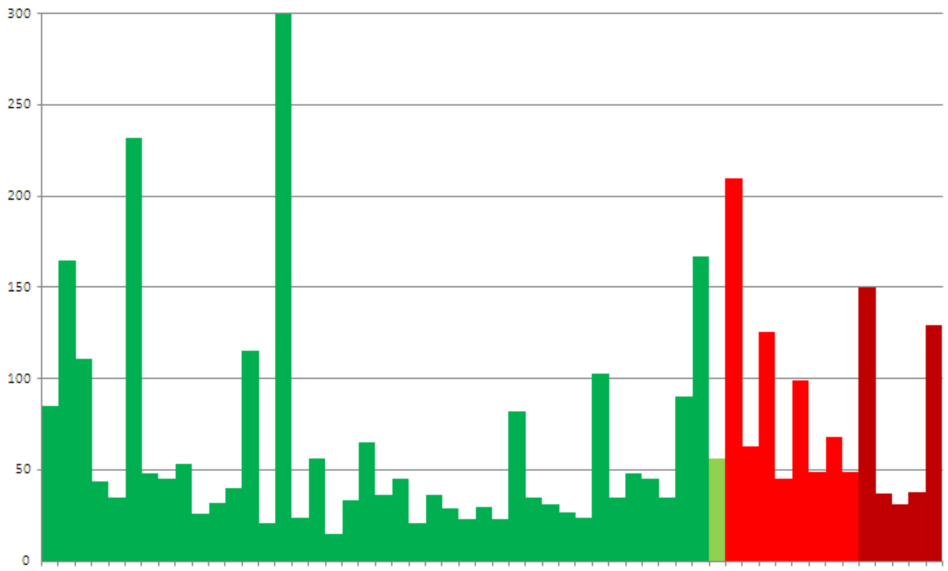


Figure 5.20: *Debugging times using the procedural debugger*

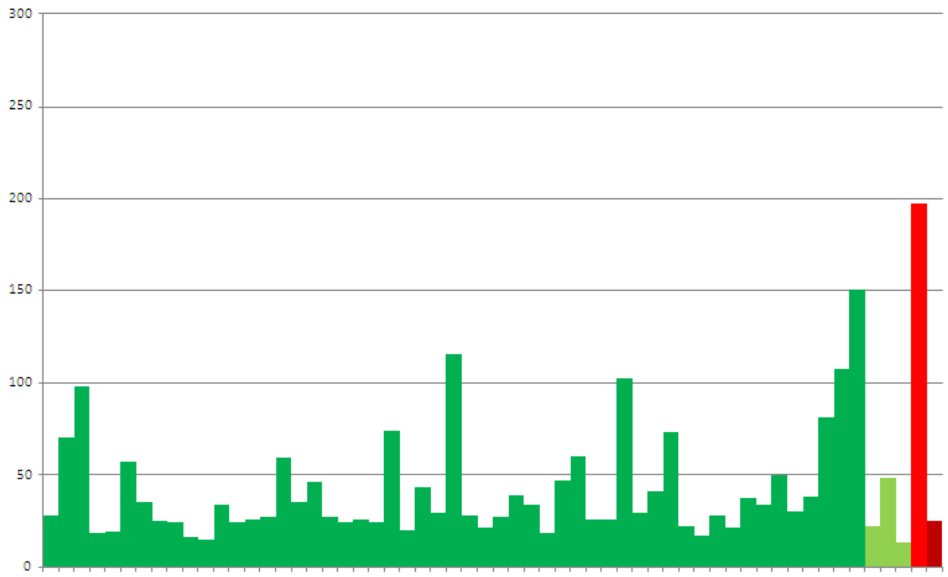


Figure 5.21: *Debugging times using the Explorative Debugger*

little or no browsing in the rule base was necessary. In fact in 6 cases the user identified the fault without any navigation at all (however, one of these cases was only *almost correct*; it identified the right atom but not the right term). This stands in contrast to the use of the procedural debugger, where the users almost always had to observe the inference engine for many steps.

A further observation was that with either debugger the **no-result-case** turned out to be often simpler to debug than cases with false results. For the no-result-case the Explorative Debugger often showed a partial or mutated prooftree with one red node that allowed almost instant identification of the fault location. For a wrong result also a prooftree was given, but the user did not have the one red node that he or she could use as starting point. When debugging a no-result-case with a procedural debugger the user could concentrate her attention on failed goals; quickly stepping through the goals succeeding. This too, was not possible in the cases where the user had to identify the reason for a wrong result.

The failure of rules to conclude the expected result based on the **interaction between multiple atoms** proved to be difficult to diagnose. Often, at least with the error seeding used in this experiment, the fault caused one rule atom to become unsatisfiable; there was absolutely no binding for this atom. Such cases are easily understood by users and easy to diagnose; the Explorative Debugger even supports these cases by highlighting the unsatisfiable atoms in red. More difficult were cases, where every single body atom was satisfiable, but no variable binding satisfied all of them. These cases needed more time to diagnose, both in the Explorative and the procedural debugger. One venue for further improvements to the debuggers hence would be a better explanation for these cases, for example based on the largest satisfiable groups of atoms.

A final observation was that the **effectiveness of the mutated prooftrees** largely depends on whether the kind of faults encountered are 'compatible' to the similarity function used. In the evaluation the inverse of the edit distance was used as similarity measure; mutated prooftrees built with this similarity function proved to be only good at finding faults based on the adding or removing of characters; i.e. specific kinds of typos. It can be expected that changing the similarity function to give high similarity values to the kinds of faults created by other seeding methods (such as changing an URI to a literal or even replacing a term by a synonym) would have made the mutated prooftrees even more effective. For the practical application of mutated prooftrees this means that the similarity function must be fine tuned to detect actual fault patterns, e.g. for example giving high similarity values to mutations that replaces a term with similar term in a different namespace or high values for the replacement of a class with its superclass.

5.3.5 Evaluation Discussion

The evaluation showed a significant decrease in time while the accuracy increased; with this it showed the potential of Explorative debugging to outperform state of the art debuggers under specific circumstances. However, there are also many questions with respect to the approach presented in this chapter that cannot be answered without additional evaluations. The most important of these questions will be shortly explained in the following:

- **Larger rule bases:** The number of rules in the evaluation rule bases was very small. It remains an open question how the approaches compare for larger rule bases.
- **Contribution of different Components:** It remains unclear how important each of the different components of Explorative Debugging was; e.g. it may theoretically be the case that Explorative Debugging would show exactly the same performance without partial prooftrees. This question is particularly acute for the relative importance of partial and mutated prooftrees, because each of these is relatively expensive to implement.
- **Fault seeding:** The evaluation experiment used fault seeding based on qualitative observations from rule base development experiments. No data is available on how well these reflect the actual faults encountered. An experiment using actual faults in real rule bases could have a much higher predictive ability.
- **More users:** Lastly the evaluation is based only on the use of these debuggers by two persons. A repetition with more users would also be very desirable.

5.4 Prior Work

Throughout the history of rule based systems a number of different paradigms have emerged to tackle the debugging challenge. An overview of the debugging approaches and their relationship is shown in figure 5.4.

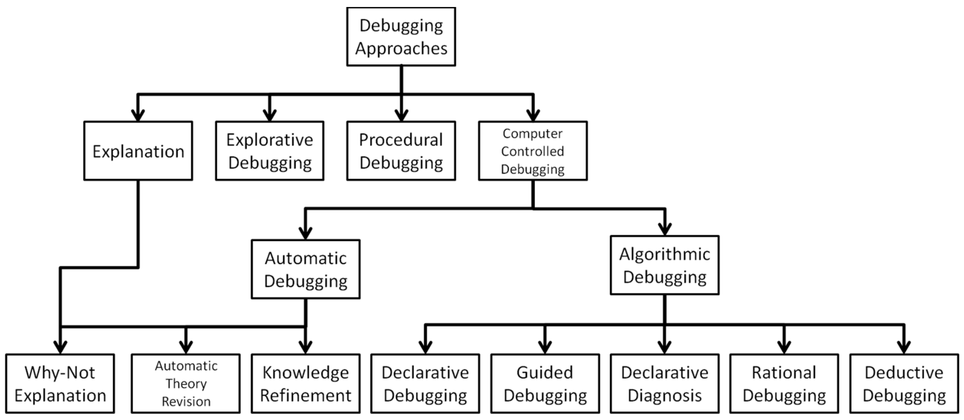


Figure 5.22: *A schematic overview of debugging paradigms*

The main characteristics of the approaches in the figure are the following:

- **Explanation** systems generate human understandable representations for why a particular result was returned. Explanation systems were developed initially not for debugging but rather to increase user acceptance of a rule based system by making its decision understandable. Explanation approaches are described below in section 5.4.2.

- The novel paradigm of **Explorative Debugging** enables the user to explore the execution of the rule base at the declarative level; this debugging paradigm has been introduced in the main body of this chapter.

- **Procedural Debugging** is well known from the world of object oriented and procedural programming languages; it uses the concepts of breakpoints, stepping and program state exploration to enable the user to observe the behavior of the inference engine. It is described in section 5.4.1.

- **Computer Controlled Debugging** is a broad term for approaches that strive to isolate the user from the actual execution of the rule base. With these approaches the user is no longer in control of an explorative process to find the fault; rather this process is controlled by the computer. The user is only presented with the results and occasionally has to answer questions posed by the system.

- **Automatic Debugging** approaches automatically identify the bug causing fault from a large number of candidates. Some systems may also return a small number of likely faults. Some automatic debugging systems also automatically change the rule base to remove the bug.

- **In Algorithmic Debugging** the execution of the computer program is decomposed into a tree of sub-computations. Some oracle (the specification, another system or the user) is then used to check the results of sub-computations; a computation that returns a wrong result from correct input is the location of a fault. Algorithmic Debugging is described in detail in a section below. Declarative Debugging, Guided Debugging, Declarative Diagnosis, Rational Debugging and Deductive Debugging are different names for essentially the same concept.

- **Why-Not Explanations** are special explanations that can also explain why a particular result was not returned. Why-Not Explanations exist in two variants, in one they rely on the user to ask consecutive questions to find the root cause, in the second the system automatically identifies the root cause and presents an explanation for that. Why-Not Explanations are the topic of a later section.

- **Automatic Theory Revision** and **Knowledge Refinement** are automatic debugging approaches that also change the rule base to remove the fault. Approaches called Automatic Theory Revision have their roots in machine learning and require relatively large sets of test data, Knowledge Refinement approaches require less data but often more user involvement, i.e. they may require a user to select one of a number of possible changes to the rule base. These approaches are jointly described in a later section.

Of these approaches the most used is clearly procedural debugging, our survey found more than 70% or rule base development projects using either a graphical or a textual procedural debugger (see figure 5.4). 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 industry strength implementations. The following sections will give a short overview of these approaches, with more space devoted to those in practical use.

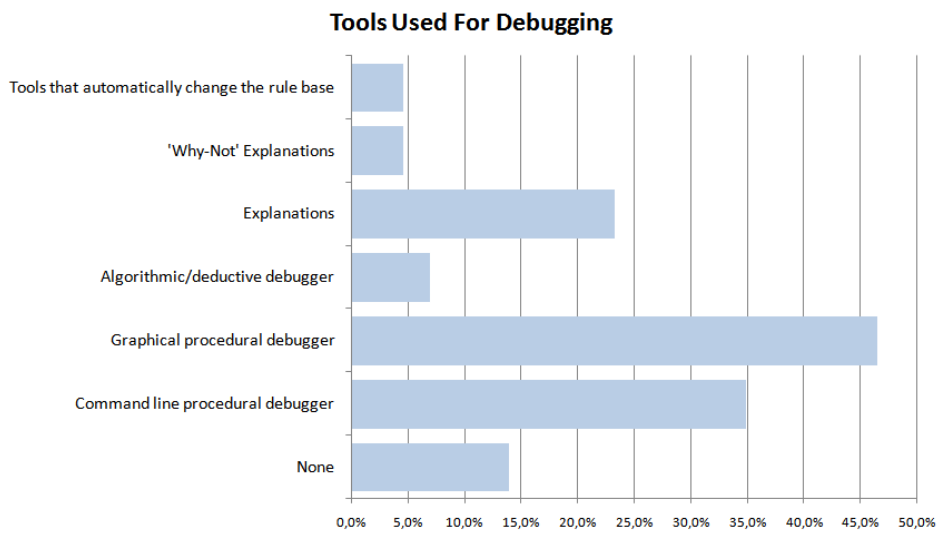


Figure 5.23: *Tools used for debugging - percentage of respondents that stated to use a particular tool for debugging during the development of a rule base*

5.4.1 Procedural Debugging

Procedural Debugging is by far the most widely used debugging paradigm for rule based systems. Debuggers following this paradigm are well known from the world of procedural and object oriented programming languages. A procedural debugger offers a way to indicate a rule/rule part where the program execution has to stop and has to wait for commands from the user; this location is called breakpoint or spypoint. The user can then examine the current state of the inference engine and give commands to execute a number of the successive instructions, to step or creep. However, a rule base strives to be a declarative representation of knowledge and does not directly define an order of execution - hence the order of debugging is based on the evaluation strategy of the inference engine.

Consider the following rule base as example, note the fault introduced to the fact in the last line.

$motherOf(A, X) \leftarrow female(A), child(A, X)$
$fatherOf(A, X) \leftarrow male(A), child(A, X)$
$parent(Y) \leftarrow fatherOf(Y, X)$
$parent(Y) \leftarrow motherOf(Y, X)$
$male(Peter)$
$child(Mike, Peter)$ # fault - should be child(Peter,Mike)

To this rule base the user poses the query $\leftarrow parent(Y)$ and is surprised to get no result. For this example it is further assumed that the rule base is evaluated with a backward chaining inference engine and that the user suspects the rule $parent(Y) \leftarrow fatherOf(Y, X)$ to be faulty and hence has placed a breakpoint on this rule. The debugging process with a procedural debugger then could be the following:

1. The inference engine starts to try to find results for the query $\leftarrow parent(Y)$. As one of the first things it will find the rule $parent(Y) \leftarrow fatherOf(Y, X)$ and determine that this rule could help in finding a result. The inference engine designates $fatherOf(Y, X)$ as the next goal to be proven and then suspends itself because a breakpoint is associated with this rule.

2. The user is presented with information about the current state of the inference engine (what rule the inference engine is examining, what is the goal it is trying to prove, what results have already been found etc.) and is given a choice on how to proceed. The choices are usually to abort the inference process, to let the inference engine resume until it hits the next breakpoint, to look at the next steps of the inference engine or to jump to next condition in, or the end of the current rule. Some advanced systems also allow going backwards in time and looking at the previous steps of the inference engine. In this example the user decides to follow the stepwise execution of the inference engine.

3. The inference engine finds the rule $fatherOf(A, X) \leftarrow male(A), child(A, X)$, designates $male(A)$ as the next goal to be proven and suspends.

4. The user is presented with information about the state of the inference engine and is again given a choice on how to proceed. The user decides to jump to the next goal of the rule.

5. The inference engine successfully proves male(A), binding the variable A to Peter. It then designates $child(Peter, X)$ as the next goal to be proven and suspends.

6. The user looks at the presented program state and decides to let the inference engine step.

7. The inference engine fails to prove $child(Peter, X)$ and suspends.

8. The user is informed about the failed goal, realizes the fault and ends the execution of the program.

The exact nature of the steps, the terminology used, the way in which the inference engine's state is conveyed and the overall user interaction varies greatly between different procedural debuggers, however, they all share the three defining properties: 1) they are based on the procedural nature of the inference engine 2) they offer a way to control the stepwise execution of the inference engine 3) the state of the inference engine can be examined when it is suspended.

The most common terminology used is based on the Byrd box mode [28] This model of execution is used to give a uniform procedural view on the execution of a logic program. In this model each predicate is understood as a procedure or box. A box is understood to have four ports, the program control is said to enter or leave through one of them. The ports are:

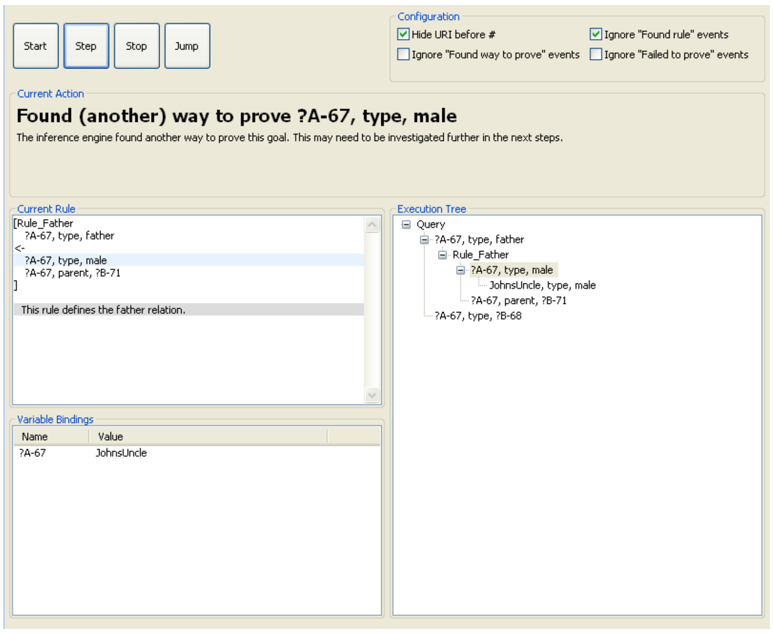
- **Call:** The call port is used when a procedure is called the first time, the first time a goal is attempted. The control flows into a box through the call port.

- **Succeed:** The succeed port is used when control flows out of a box after a procedure executed successfully, i.e. a rule could be found such that its head unifies with the goal and all its subgoals could be proven.

- **Redo:** The redo port is only reached when a procedure has been executed succesfully before and some subsequent goal has failed. In such a case the procedure is reentered to retrieve other variable bindings that could be used to attempt the goals that failed; i.e. the control flow re-enters through the redo port in cases where the Prolog interpreter backtracks after a failed goal.

- **Fail:** Control flows out of a procedure through the Fail port in the case when (a) control entered through the *call* port but no rule's head unifies with the current goal (b) a rule is found whose head unifies with the current goal but its subgoals cannot be proven or (c) the procedure was entered through the *redo* port and no new variable bindings could be found.

Some systems define a fifth, an exception port. Control flows out of this port in case of an error.



Configuration

[✓] Hide URI before # [✓] Ignore "Found rule" events
[ ] Ignore "Found way to prove" events [ ] Ignore "Failed to prove" events

Current Action

Found (another) way to prove ?A-67, type, male

The inference engine found another way to prove this goal. This may need to be investigated further in the next steps.

Current Rule

```
[Rule_Father
   ?A-67, type, father
<-
   ?A-67, type, male
   ?A-67, parent, ?B-71
]

 This rule defines the father relation.
```

Variable Bindings

Name	Value
?A-67	JohnsUncle

Execution Tree

```
⊟ Query
   ⊟ ?A-67, type, father
      ⊟ Rule_Father
         ⊟ ?A-67, type, male
              JohnsUncle, type, male
           ?A-67, parent, ?B-71
     ?A-67, type, ?B-68
```

Figure 5.24: *Screenshot of a conventional procedural debugger implemented as part of this thesis*

The user interface of a modern procedural debugger with the typical user interface elements is shown in figure 5.4.1

- On the top left the buttons 'Start', 'Step', 'Stop' and 'Jump' enable the user to control the inference engine.
- In the top middle, in the 'Current Action' box, a short explanation of the last action of the inference engine is given.
- The 'Current Rule' box shows the rule currently processed by the inference engine. The current condition (A is of type male) is highlighted in blue. A rule comment is shown with a gray background.
- The variable bindings box shows the variables and the constants they are currently bound to, in this case the rule engine has already bound A to JohnsUncle.
- On the bottom right the execution tree shows an overview of the search process by the inference engine.

Procedural debuggers are available as purely textual tracers [170, 169] and with a simple graphical user interface [160, 170]. The most sophisticated graphical user interface of a procedural debugger was created as part of the Transparent Prolog Machine project [56, 55] that displayed even large inference processes in concise 'AORTA' (And/Or TRee Augmented) diagrams [26, 126, 53]. Modern business rule management systems (e.g. [122]) include sophisticated procedural debuggers that do not include this kind of visualizations but that are integrated with graphical rule editors.

5.4.2 Explanations

Explanations are abstractions of the program trace that aim to explain the program execution to a user. Explanations can be graphical, textual or both. Some explanation systems generate and display one explanation after the program execution, some offer an interface for the user to investigate the inference process, others stipulate a dialog with the user. Explanations are used to build confidence in the results of expert systems, to facilitate better understanding of rule based systems, to teach problem solving and for debugging.

Explanations are most commonly based on the prooftree, a representation of the inference process that led to a result. A prooftree contains the rules and facts and connections between them that were used to derive the result. A prooftree can directly form the input for explanation visualizations or it can be used as a basis to derive more refined explanations. Prooftrees are created by the inference engine during the search for a solution, they are, however, relatively independent of the actual implementation of the inference engine. The same prooftree can be created by a forward chaining, backward chaining or any inference algorithm. For some rule languages (e.g. Datalog) prooftrees can be defined to depend only on the rules and facts and not on the inference engine.

Consider the example from the previous section (after the software fault has been removed).

$motherOf(A, X) \leftarrow female(A), child(A, X)$ # Rule motherOf
$fatherOf(A, X) \leftarrow male(A), child(A, X)$ # Rule fatherOf
$parent(Y) \leftarrow fatherOf(Y, X)$ # Rule parent_1
$parent(Y) \leftarrow motherOf(Y, X)$ # Rule parent_2
$male(Peter)$
$child(Peter, Mike)$

For the $query \leftarrow parent(Y)$ the system now returns the result $Y = Peter$ and the prooftree for this result is shown in figure 5.25.

At the top the root of the prooftree is formed by the Result $Y = Peter$. This node is connected to a node representing the application of the rule

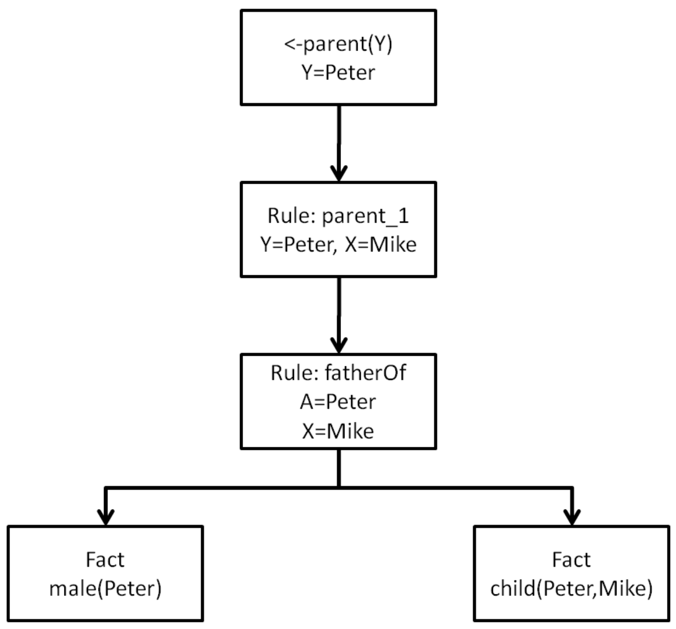


Figure 5.25: *Simple schematic prooftree for the example in the section explanations*

$parent\_1$ with the variable Y bound to $Peter$ and X bound to $Mike$. This in turn depended on an application of the rule $fatherOf$, which used the two facts $male(Peter)$ and $child(Peter, Mike)$ at the bottom of the prooftree.

A prooftree is often augmented with natural language templates that allow to create readable representations of rule applications. For the example above these explanation templates might look like the following:

```
motherOf: %A is the mother of %X, because she is female and
   has %X as a child.
fatherOf: %A is the father of %X, because he is male has and
   has %X as a child.
parent_1: %Y is a parent, because he is the father of %X.
parent_2: %Y is a parent, because she is the mother of %X.

child(X,Y): %Y is the child of %X.
male(X)    : %X is male.
parent(X)  : %X is a parent.
```

Such templates are created by the developers of the rule base and may also contain additional information such as when and by whom a rule was created or justifications in the form of documents that were used during the creation of the rule. With these templates a readable explanation for the results can be created, such as:

```
Peter is a parent
  Peter is a parent, because he is the father of Mike
    Peter is the father of Mike, because he is male and
    has Mike as a child.
      Peter is male
     Mike is the child of Peter
```

As is already evident from the small example, ensuring readability and conciseness of these approaches is challenging and often requires additional steps to simplify the prooftree and the explanations.

Explanations following this principle are used in some business rule systems [85] and many other rule systems (e.g. [108, 61, 62]). A recent evaluation of the quality of the created explanations can be found in [61].

Historically, explanations have been used in expert systems since the pioneering Mycin system [153, 152, 25], a system for the diagnosis and treatment of bacterial infections. Mycin was meant to work as an assistant to physicians in the diagnosis and treatment of bacterial infections. Early versions of Mycin only offered rule traces as explanations for results, but sophisticated explanation algorithms where developed in the Teiresias [46, 47] project and then added to the Mycin system.

The user interacts with Mycin in the form of a textual dialog. A typical dialog starts with the physician giving some facts about the patient in a form of constrained English. Mycin then tries to infer a diagnosis and recommend a treatment with the use of a production rule system. The system asks questions like: "What is the protein value in the CSF [Cerebro-Spinal Fluid]?" when it needs more evidence to reach a conclusion. At all stages during the interaction the user can demand explanations from the expert system. She can ask questions like "Why is it important to know the protein value in CSF", "Did you consider that the patient had not received steroids?" or "How did you know that the patient had not received steroids". Mycin attempts to match the question of the user against a number of predefined patterns. After a suitable pattern has been found, an algorithm for this type of question generates an answer. The answers are generated by looking directly at the rules and a trace of the inferences made: "why" question would be answered by giving the conclusion of the rule(s) that use(s) this evidence. The "did you consider" question by finding all rules that could use this fact and saying whether they were used and if not, which rule part prevented them from being used. The system answered "How" questions by finding and displaying rules that concluded a fact. Although mainly used to explain and build confidence in the inference process, explanations where also used as a high level debug tool [166]. A longer description of Mycin's explanation facilities can be found in [149] and [153].

The success of the Mycin system [175] led to many follow up projects, of particular interest is the Guidon [37, 38] project that tried to leverage the rules created by experts for Mycin for teaching purposes. Other important work includes the Knight [98] system renowned for its ability to generate polished multi-sentence and multi-paragraph explanations. The Texplan [105, 106] system that gives the user the chance to influence the explanations by asking for elaboration or stating disbelieve. Or finally the older XPlain [159] system that pioneered (together with Guidon) the use of explicit domain knowledge in explanation generation.

Explanation systems have been and are used successfully in many systems; these systems are important in particular to build confidence in the results from the expert systems. The utility of explanations for debugging, however, is limited because these approaches (except for the Why-Not Approaches described below) can only be used when there is a result, not in cases where a fault prevents a query from having any result (as, in the example in the section on procedural debugging).

5.4.3 Why-Not Explanations

Of particular interest for debugging purposes are explanation systems that can answer "Why Not" questions such as "Why was there no result?" or "Why didn't the result equal the one that I know to be correct?" Only a small number of explanations systems can answer this kind of questions. The Teiresias system described above allowed for simple why-not questions, again finding rules that conclude the user expected result and returning that parts of the rule body that prevented these rules from firing. The task of providing why-not explanations in this system was simplified by the very limited expression power of its rules. Why-not explanations exists in two distinct variants: a manual variant where a why-not explanation is only generated for the current rule or query and an automatic variant that tries to automatically identify the ultimate cause. In this section an example for the manual variant is given first followed by one of the automatic variant, both using a variant of the already familiar rule base and the query $\leftarrow parent(Y)$.

$motherOf(A, X) \leftarrow female(A), child(A, X)$ # Rule motherOf

$fatherOf(A, X) \leftarrow male(A), child(A, X)$ # Rule fatherOf

$parent(Y) \leftarrow fatherOf(Y, X)$ # Rule parent_1

$parent(Y) \leftarrow motherOf(Y, X)$ # Rule parent_2

$child(Peter, Mike)$

# Error, note that male(Peter) is missing

A debugging session with a manual Why-Not debugging system would start with the system giving an explanation for the failure of the query to find a result (this example is inspired by the capabilities of the system described in [20].

```
I can’t prove that there is a Y such that parent(Y)
because
  Rule [parent_1] is not applicable:
    there is no X such that
       fatherOf(Y,X)    <details?>

  Rule [parent_2] is not applicable:
    there is no X such that
      motherOf(Y,X)    <details?>
```

Using her knowledge about the system the user would then choose to see details about fatherOf(Y,X), getting an explanation pinpointing the location of the problem:

```
I can’ prove that there are X,Y such that fatherOf(Y,X)
because
  Rule [fatherOf] is not applicable:
    there is no Y such that
      male(Y)
```

Unlike this simple example, however, real-life application of this kind of explanation requires sophisticated algorithms to hide details that are unlike to interest the user and also to make it possible to keep an overview of large rule bases. This kind of system also requires some intervention of the user, possibly too much for end-user facing applications. Why-Not applications of this kind have been known for a long time, but are also not currently used in commercial systems. Besides the Teiresias and the Rewerse system [20] another example is Que [104], which applies these techniques to a forward chaining inference engine. Automatic why-not explanations strive to make debugging even simpler by automatically identifying the fault that caused a bug or by automatically producing a detailed explanation for why the rule base has not been able to conclude the expected result. However, since these systems cannot possibly have complete knowledge about the domain of the system and the intention of the programmers, the identification of the ultimate cause is done heuristically. In the example rule base described above, possible faults that explain the failure of the rule base to find any Y such that $parent(Y)$ are, for example:

```
the fact male(Peter) missing
rule fatherOf wrongly contains the condition male(A)
rule motherOf wrongly contains the condition female(A)
child(Peter,Mike) should be child(Mike,Peter) and
    the fact male(Mike) is missing
```

```
rule parent_1 should have the condition child(X,Y)
    and not fatherOf(Y,X)
  rule parent_2 should have the condition child(X,Y)
    and not motherOf(Y,X)
the fact fatherOf(Peter,Mike) is missing
the fact parent(Peter) is missing
the fact parent(Michael) is missing
...
```

This list is not exhaustive, in fact, even for this very simple rule base created in a very simple rule language there are infinitely many candidate faults. Hence, automatic why-not systems must choose one (or a few) of these infinite many candidate faults for presentation to the user. Implicitly this selection is based on the competent programmer hypothesis [51]. It states that, while unable to create flawless programs, programmers usually create programs that are "nearly perfect" and that syntactically small changes are sufficient to correct them. Applying this idea to the search problem the system can look for the smallest change that would make a query succeed and report this as the most likely problem. In a nutshell, automatic why-not explanation systems change the inference engine to also look for proofs that almost succeed - that use a good part of the rule base and that with just a small change would return a result. These systems build on the theory of abduction [128, 89] - a reasoning paradigm that tries to identify the best hypotheses that, when true, would explain a phenomenon.

The most important of these systems is the why-not system by Chalupsky and Russ [32] that also explored the applicability of these techniques to large rule bases. A very recent one is the system by Becker et al [8] that uses these techniques to explain why a system denies access to a resource based on policies formalized as rules. Diamod [111], Diva [44] and DeBrief [87] are further systems that can answer "Why Not questions". Automatic why-not debugging systems seem promising, but have not yet made the leap to practical application. Possible reasons are the difficulty in implementing these systems, the potentially high computational complexity and the still open question of whether these systems can reliably identify real life errors.

5.4.4 Knowledge Refinement and Automatic Theory Revision

Automatic knowledge refinement and automatic theory revision systems take as input a knowledge base or "theory" and a set of test cases and their correct results. Some of the test cases fail with the given knowledge base. These systems then try to change the knowledge base until all test cases succeed. Automatic knowledge refinement systems usually employ an iterative control structure: identify a set of possible repairs, evaluate them on all test cases and apply the most promising refinement. This is repeated until all test cases succeed or the system fails to generate any refinement that improves the performance of the knowledge base.

Historically [41], automatic knowledge refinement systems have been part of the knowledge acquisition community. These systems deal with many different problem solving approaches and representations. They often require some intervention by a human expert. In contrast automatic theory revision systems have been created by the machine learning community; they require many test cases but usually no human intervention.

The Krust system [40, 41] is a recent automated refinement system for knowledge based systems. Starting from a failed test case and the correct solution the system classifies rules into categories such as "error causing" (a rule that fires and whose conclusion contradicts the correct solution) or "target rule, NoCanfire" (a rule that would conclude the correct solution but whose antecedents are not satisfied). The system then generates a large number of possible refinements by making small changes to the knowledge base based on the class of a rule (for example dropping one clause from the antecedent of a "target rule, NoCan fire"). The number of possible refinements is then reduced through the use of heuristics such as "prefer less changes", the ability of refined knowledge bases to correctly process other stored test cases or meta-knowledge about rule quality. The best scoring refinements are presented to the user. Other famous knowledge refinement systems are Seek [67] and Odysseus [171]. Probably the first automatic theory refinement system was the poker playing program from Waterman [167]; other examples are Ether [125], Forte [112] and Audrey II [172]. Multi-strategy revision systems use several learning techniques and knowledge sources. Clint [136], Why [147] and Mobal [113] are examples for multi-strategy revision systems.

Related, although seldom used for debugging, are rule induction and knowledge assimilation approaches. Rule induction [115] focuses on learning rules from data without consideration for rules and facts already present. Knowledge assimilation [57] is the process of incorporating updated data into a knowledge base while taking into account the content and integrity constraints of the knowledge base. In contrast to theory revision, automatic knowledge refinement and rule induction focus on adding/removing facts and not on changing rules. Knowledge assimilation relates to debugging in the following way: the correct result of a failed test is understood as an update request, knowledge assimilation techniques are then used to identify facts that could be added to the knowledge base to make this test succeed.

Some people argue that automatically learned and refined rules are the only way in which rule bases can find wider practical use, however, despite decades of research these approaches are currently confined to mostly academia. These approaches still have to prove that they can be used efficiently and reliably to maintain rule bases. There is also the problem that many current applications of rules focus on reliability and traceability, i.e. these applications are realized with rules solely to ensure that business processes are reliably executed and that decision can be justified - both properties that are risked when rules are learned. Another problem is that these approaches are relatively hard to implement and computationally expensive.

5.4.5 Algorithmic Debugging

Algorithmic debugging describes debugging techniques that automatically divide the program into hierarchically ordered part's, each part having a number of subparts that are used in this parts computation. Each part has some identifiable result. These approaches then try to identify the part(s) that create a wrong result even though its subparts return correct results.

Ehud Shapiro [151] described the first algorithmic debugger in his PhD thesis. The system creates a computation tree representing the computation of the program. The nodes in the tree represent procedures, rules or other small programming parts and their results. The children of a node are other programming parts whose results were instrumental in this node's computation. The root node represents the entire program and its result. Using the user and the specification as an oracle the system identifies the node in the computation tree that returned a wrong result from correct inputs, i.e. the program uses an algorithm to identify a suitable candidate node, asks the user "is the result of this node correct?" and repeats this until the faulty node is found. The system tries to minimize the number of questions asked to the user.

As an example consider the well known rule base, similar to the one in the previous section but with an additional fact $male(Mike)$.

$motherOf(A, X) \leftarrow female(A), child(A, X)$ # Rule motherOf
$fatherOf(A, X) \leftarrow male(A), child(X, A)$ # Rule fatherOf;
error, should be child(A,X)
$parent(Y) \leftarrow fatherOf(Y, X)$ # Rule parent_1
$parent(Y) \leftarrow motherOf(Y, X)$ # Rule parent_2
$male(Peter)$
$child(Peter, Mike)$

In the following, an example for a debugging session with an algorithm for the query $\leftarrow parent(Y)$ is shown. In each step the system presents the user with the input for a program part and its output. The user replies with yes or no indicating whether this program part performed according to her expectations. Note that the fault introduced above means that the query returns the false result $parent(Mike)$ instead of the correct $parent(Peter)$.

```
parent_1 (in: Y, out:  Y=Mike)
user> no

fatherOf (in: Y,X out: Y=Mike, X=Peter)
user> no

male (in Y, out:Y=Mike)
user> yes

child (in X,Y out: X=Peter,Y=Mike)
user> yes

A fault has been identified in the body of rule fatherOf.
```

A large body of research into algorithmic debugging exists, although often using a different terminology. Algorithmic debugging has also been called declarative debugging [155], declarative diagnosis [118], guided debugging [22], rational debugging [129] and deductive debugging [52]. Discussion of graphical user interfaces for algorithmic debugging can be found in [168] and a collection of useful references related to algorithmic debugging is contained in [53].

Algorithmic debugging has been heralded as a promising debugging approach because it truly frees the user from understanding complex interaction in the program and because it can seamlessly use any (even partial) specification to reduce the number of questions asked to the user. However, at the same time algorithmic debugging still asks a large number of questions, some of which may be very hard to answer. Further it does not facilitate learning of the user about the program and cannot easily profit from the user's intuition and background knowledge that may aid in finding a fault. Algorithmic debuggers have not, so far, found widespread adoption and the author is not aware of any commercial rule based system including such a debugger.

5.4.6 Synthesis - Explorative Debugging Compared to Prior Work

As a debugging paradigm, Explorative Debugging aims to fill a void between approaches that require the developer to understand the inference engine (procedural debugging) and, on the other hand, approaches that are at abstraction levels considerably above those of rules (explanation and computer controlled debugging). It is the first systematic attempt to debug rule bases at the **same level** as the semantics of the rules.

Procedural debugging forces the developer to learn about the inner workings of the inference engine. In this way one possible advantage of rule based systems - that the developers do not need to worry about the *how* of the computation - is lost. An additional problem is that the inner workings of inference engines can actually be very complex and even harder to understand than a traditional backward chaining inference engine.

On the other end of the spectrum explanation and computer controlled debugging approaches try to shield the developer from the details of the debugging process; for example by delivering one shot explanations or through debugging dialogs. However, there are a number of high level problems with these approaches:

- These approaches always risk abstracting not only from unecesssary details but also from the actual fault.
- Some of these approaches (in particular Algorithmic Debugging) also get into the way of the developer; prevent her from using all her knowledge and intuition to quickly find a fault.
- The performance of these approaches in the automatic identification of faults has never proven to be good enough to debug the overwhelming majority of bugs in a program; even in the much more mature world of procedural and object oriented programming no such tools have emerged. For this reason these tools can never be the sole debugging support - at the very least, they need the backup of a manual debugging tool.

In addition to these abstract contributions, the Explorative Debugger also introduces new ways of capturing the static and dynamic structure of a rule base (in particular the mutation extended depends-on graph and the mutation extended prooftree).

5.5 Conclusions and Future Work

Explorative Debugging is a new debugging paradigm for logic programming that enables developers to explore the execution of logic program without needing to worry about the evaluation strategy of the inference engine. As such it aims to fill a void between the procedural debugging approaches on one hand and Algorithmic Debugging and explanation approaches on the other.

This chapter presented properties of logic programs that can be used to explore the execution of a logic program at the declarative level. This work adds to the state of the art in debugging of rule bases in four points: (1) the notion of Explorative Debugging, (2) the comprehensive collection of declarative properties of rule bases useful for debugging (3) the notion of the mutation extended dependency graph and (4) the notion and the implementation of the mutated prooftree (in particular in a way that allows to use arbitrary similarity measures such as taxonomic similarity).

The presented implementation of the F-logic debugger was the first ever graphical debugger for this language and is still the most powerful. The Trie! package is the only debugger for the presented rule language, the only open source implementation of partial prooftrees and the only implementation of the mutated prooftree concept.

This chapter further presented an experiment comparing Explorative Debugging to the state of the art procedural debugging and showing a significant improvement in the time needed to identify faults while the accuracy increased.

The work presented in this chapter opens up numerous venues for further research:

- To arrive at more definitive answers about the suitability of different debugging approaches it would be desirable to extend the evaluation to larger rule bases, examine the importance of the different components of Explorative Debugging and use faults not artificially created but that occurred in real rule base development projects.
- Extending the Trie! implementation to cover a broader rule language, in particular to also support built-ins and tabling to allow rule cycles. A related research thread would be the introduction of partial and mutated prooftrees into the Inference Explorer for F-logic.
- Another very interesting research thread would be the adaptation of the algorithms to create partial and mutated prooftrees to perform best-first instead of depth-first search; i.e. to create inference engines that only start to explore mutations and partial prooftrees af-

ter (at least the simple) chances to succeed without these have been exhausted.

- Finally the mutation extended prooftree introduces a fairly general model for conclusions that could almost be reached. This notion of near conclusion is potentially useful for areas outside of debugging; i.e. domains with noisy and faulty rule bases where some false conclusions are acceptable.

Chapter 6

Anomaly Detection

Anomaly detection is the automatic examination of a rule base in order to find symptoms of probable fault; to identify parts of a rule base that are in need of a developer's attention. Anomaly detection is a conceptual framework for looking at some automatic techniques for (mostly) static analysis of a rule base under development.

Static analysis was identified in the design and analysis of this thesis (see chapter 3) as one of the transitions to investigate. This was based on the challenge caused by the absence of formal specifications in development projects using agile methods and on the novel opportunities afforded by terms described in ontologies.

Static analysis means techniques for the examination of a program without running it; it is performed for two related purposes: 1) to find faults that need to be removed and 2) to obtain evidence for the conformance of a program to (some part of) its requirements; i.e. for program verification. These two purposes are related, because many of the static analysis techniques used for verification also provide information that can be used to find and remove faults.

At a high level, five different (but overlapping) clusters of static analysis methods can be identified:

- **Code Review:** The manual inspection of a program by its developer, a different developer, a domain expert or an external specialist.
- **Bug Patterns:** The automatic, usually heuristic, identification of patterns in the program that can be identified as faults or that are often seen as a consequence of a fault. Examples are rules that can never fire, duplicate rules or variables that are never read.

- **Formal Methods:** The use of reasoning techniques to formally proof important properties of a program.

- **Software Metrics:** Measures of some property of a software or parts thereof. Examples for software metrics are lines of code, coupling, the number of rules or the size of the vocabulary of a rule base. Most software metrics are computed without executing the program (test coverage metrics are one important exception).

- **Type Checking:** Type checking is a specific way to identify useless and potentially problematic parts of a computer program. Type checking often depends on particular elements of the programming language that enable developers to specify the intended type for entities in the program. In addition to their use in static analysis, many type systems also affect the runtime of the program, e.g. they enable to choose between similarly named, but unequally typed methods (i.e. overloaded methods). Types are described in more detail in section 6.2.

Note that these clusters of methods overlap, e.g. software metrics and bug patterns can guide the review process and bug pattern detection can utilize type information for more effective fault identification. Also note that the static analysis is often understood to only encompass automatic techniques, in this way excluding code review.

In the context of the work presented here, the decision was made to put the notion of *anomaly* at the core of the static analysis, since it is a very flexible notion that can encompass a large group of static analysis methods. An anomaly is a notable, distinctive or unusual part or property of a rule base; it points the developer to some part or property that requires attention. Anomalies encompass bug patterns, many problems with types and can also utilize software metrics.

After this introduction, the notion of anomalies is further explored in two sections, first introducing the anomalies related to the type systems and then those related to bug patterns and metrics. The subsequent section then describes the implementation of these notions. Finally a section introduces prior work before this chapter concludes.

The work described in this chapter is only related to F-logic and does not easily generalize to normal logic programs. This is caused by the fact that normal logic programs (as introduced in the chapter 2.1), lack a type system comparable to F-logic and also that many of the anomaly heuristics introduced use specifics of types or F-logic programs that are not found in normal logic programs.

6.1 Anomalies and Anomaly Heuristics

Anomalies are notable, distinctive or unusual parts or properties in a rule base that are, with a high probability the result of a fault. Anomalies are used to focus the developer's attention on some part of the knowledge base that is likely to contain a fault.

Anomalies can be computed using any kind of method or tool, such as the detection of bug patterns, type checking or the computation of software metrics. The methods used to compute anomalies are called **anomaly heuristics**[1]. Every anomaly heuristic described below detects one type of anomalies.

In order to ensure portability and following the seminal work of Preece and Shinghal [133], all anomaly heuristics are realized in F-logic itself.

Ideally each anomaly would directly correspond to one fault, but ensuring this is generally impossible and many anomalies are false positives, i.e. are not actually a symptom of a fault in the rule base. To allow developers to focus on the more serious anomalies a ranking of anomalies was introduced: each anomaly is classified as either *information*, *warning* or *error*.

For better organization anomalies are presented here in two large groups: those related to the F-logic type system and those that are not. Both types of anomalies are now described in one section each.

6.2 Type Checking based Anomalies for F-logic

In general types attach meaning to a collection of bits in a computer system, e.g. they determine which bits are executed as code and which are treated as an integer. Type information is used by both the computer and the developer and can facilitate program comprehension, optimization, and fault detection.

Type checking is the process of verifying that a computer program satisfies the constraints imposed by the type system. Type checking can be dynamic (i.e. be performed at runtime), static (i.e. be performed without executing the program) or a combination of both.

As described in section 2.2, F-logic includes syntactic elements for specifying types, but it does not specify a complete type system; in particular it does not specify a method for type checking. The F-logic introduction in

[1]The term 'heuristic' here is used to allude to the fact that not all anomalies are caused by faults, that anomalies can be false positives. 'Heuristic' does not say anything about the kind of algorithm used.

[92] includes a definition of type correctness and type safety as a *yardstick*. However, the defined type system is undecidable, too weak for many cases and computationally very expensive.

This section introduces the F-logic type checking system developed by the author in Project Halo. It starts first with a recapitulation of the type syntax in F-logic followed by a discussion of the *yardstick* type semantics introduced in [92]. Previous implementations of F-logic type systems are presented. Then, the developed type checking system is motivated, described and discussed.

6.2.1 Syntax and Semantics of F-logic Types

F-logic's type syntax consists of two kinds of *signature expression*, set valued signature expressions of the form `d[m@{c1,...,cn=>>e]` and scalar signature expression of the form `o[m@{b1,...,bn}->>t]`. Consider the following example rule base:

```
man::person.
woman::person.

person[mother=>woman].
person[son=>>man].
person[child@{woman}=>>person].
```

This rule base shows three examples for the application of these signature expressions. It first specifies a simple type hierarchy, stating that both `man` and `woman` are subclasses of `person`. The third line then is a scalar signature expression, specifying that the application of method `mother` to an instance of `person` yields at most a single return value of type `woman`. The line below is a set valued signature expression, defining that the method `son` applied to an instance of `person` yields zero or more instances of type `man`. Finally the last line contains a more complex set valued signature expression that states that the method `child` of `person` return zero or more `persons`, when called with a parameter of type `woman`. Note that these signature expressions can also be used both in the head and body of rules, allowing rules to change the signature of classes.

In [92] Kifer et al. define a yardstick semantics for the F-logic type system based on these signature expressions as follows:

An atom A of the form `o[m@{b1,...,bn}->t]`[2] is **covered by a signature** atom `d[m@{c1,...,cn}=>e]`, iff `o:d` and for every i=1,...,n it holds that `b1:c1`. Similarly an atom of the form `o[m@{b1,...,bn}->>t]` is covered by a signature atom `d[m@{c1,...,cn=>>e]`, iff `o:d` and for every i=1,...,n it holds that `b1:c1`. An rule base that contains only data atoms covered in this way is said to be **type correct**.

A Herbrand interpretation I of an F-logic program is a **typed Herbrand interpretation** iff:

1. every data atom in I is covered by a signature atom in I
2. for every data atom `o[m@{b1,...,bn}->t]` that is covered by a signature atom `d[m@{c1,...,cn}=>e]` it holds that `t:e` in I
3. for every data atom `o[m@{b1,...,bn}->>t]` that is covered by a signature atom `d[m@{c1,...,cn}=>>e]` it holds that `t:e` in I

The first condition is a restriction on the domain of methods; it ensures that methods are called only on objects for which they are defined. The second and third condition restrict the range of methods, they ensure that the methods satisfy the contraints imposed on them by the signature atoms. A rule base that satisfies the second and third condition is said to be **type safe**.

An F-logic program is a **well-typed program**, iff all its canonical models[3] are typed Herbrand interpretations.

Consider the following simple example:

```
man::person.
woman::person.
person[son=>>man].

Sarah:woman.
Sarah[son->>mike].
```

This example is not a well-typed program because of the statement `Sarah[son->>mike]`. This statement is covered by the signature atom `person[son=>>man]` (the rule base is type correct), however, `mike` is not known to be of type `man`, hence condition 3 from above is violated and the rule base is neither type safe nor well-typed.

As another example consider the logic program consisting of only one rule:

[2]For readability this section uses F-logic syntax, however, please recall that (as defined in section 2.2.3) this is equivalent to the normal logic atom directatt(o,m(b1,...bn),t,DefaultModule).

[3]E.g. the least or perfect Herbrand model as defined in section 2.1.

```
FORALL X,Y
    X[son->>Y]
    <-
    Y:man[father->X].
```

Intuitively this program should be detected as not-well typed, because the rule uses a class and methods not declared at all. However, because the (empty) least Herbrand model of this program contains no atoms that violate any of the constraints specified above, this program is in fact well typed in the sense described here. This example shows the weakness of this type system: it cannot identify type related faults in rules that do not fire. This weakness is particularly problematic for case where - like in Project Halo - a rule set is created to be used with changing sets of facts. In such a setting only few rules fire at any given moment and it is highly desirable to check the rules independently of the (changing) facts.

A further problem of this type system is, that checking a program for type correctness in this sense is computationally very expensive and undecidable [92].

6.2.2 Prior Implementations of F-logic Type Checking

All prior implementations of F-logic type systems have been either ad hoc syntactic approaches using procedural languages or simple implementations of the yardstick type semantics described above, even though its limitations are well understood. This section will shortly introduce the approaches taken by the three main F-logic implementations Flora, Florid and Ontobroker.

Florid [64] does not include any implementation of the type system, however, the documentation [63] includes three rules that directly implement a limited version of the yardstick type system introduced above. This implementation is limited, because it is restricted to only consider scalar data signature expression and methods with no arguments. Additionally, these rules require the prior computation of the entire least model of the program- a very costly and (in the case of infinite models) impossible prerequisite.

Flora [174] also does not automatically type check programs, but it includes a method to facilitate easier type checking by the user. This *type* method [173] can check the type correctness according to the yardstick type semantics described above; it further allows to restrict the checking of type correctness to only particular classes or modules. Type checking in Flora does not generally work for parts of the knowledge base that include built-ins or non-logical features [173] and it suffers from the general problems of the yardstick type semantics detailed above.

Ontobroker [49] does not include any facilities for type checking, however, the rule editor included with its development environment Ontostudio [124] performs static type checking. This static type checking (whose exact semantics are not published) mainly warns the user when a method is called on a variable that is not explicitly required to be of a type for which this method is specified; i.e. a warning is generated for an atom `A[son->B]` unless[4] `A:person` is also in the rule body. This type checking algorithm is implemented outside of the logic language and not portable for rules created outside of the rule editor (which only supports a subset of all possible rules). Another problem is that this type checking algorithm creates many warning for rules that would work fine, i.e. would not violate any type constraints.

6.2.3 Dynamic Type Checking Anomaly Heuristics

As part of the work described in this thesis, the dynamic type checking using the yardstick semantics was realized as two anomaly heuristics. Both take as input the module that is to be checked; thereby allowing to reduce the computational cost and making it possible to apply type checking only to the simpler modules for which it is decidable. The two anomaly heuristics are (the formal specification of the anomaly heuristics as F-logic is given in the Appendix E):

Anomaly Heuristic 1 (Instance Method Undefined) *An error is created for instances with methods that are not defined in their signature. This is a direct realization of* **type correctness***; the first of the three conditions specified for the yardstick type semantics detailed above.*

Anomaly Heuristic 2 (Method Range Type) *An error is created for methods of instances that return the wrong type. This is a direct realization of* **type safety***; the second and third of the conditions specified for the yardstick type semantics detailed above.*

These two heuristics allow a complete dynamic type checking for (parts of) a knowledge base, however, they are very expensive to compute, checking them may not terminate and they suffer from the weakness in static type checking detailed above.

[4]Assuming the signatures defined in the second example in the previous section.

6.2.4 Heuristic Static Type Checking for F-logic

Project Halo's goal was the creation of rule sets that would work for queries and facts not available at the time of their creation. In such a setting, type checking (also) needs to be static and needs to work on the rules even without them firing.

As shown above, even a full implementation of the yardstick semantic is too weak for these requirements. In general it is also not possible to determine whether any given set of rules only concludes type-correct facts. However, starting from the assumption that only data atoms (and not signature atoms and rules) are later added to the rule base, rules can be statically classified as *type incorrect*, *type safe* or *type unknown*. Consider the following example rule base:

```
/* facts */
man::person.
woman::person.

person[son=>>man].
person[father=>man].

RULE1:
  FORALL X,Y
    X[son->>Y]
      <-
    Y[father->X].

RULE2:
  FORALL X,Y
    X[son->>Y]
      <-
    Y:man[father->X] AND
    X:man.

RULE3:
  FORALL X,Y
    X[son->>Y]
      <-
    Y:man[parent->X].
```

In this example rule 1 is 'type unknown': depending on the data atoms it may conclude well typed atoms (in the case when all bindings for `Y` happen to be of type `man`) or not-well typed atoms (in the case when at least one binding for `Y` is of type `woman`). Rule 2 includes additional atoms that ensure that both `X` and `Y` are of type `man`, even with an input of data-atoms that are not well typed themselves, this rule only concludes well-typed atoms; rule 2 is type safe. Rule 3 is type incorrect - it uses a method not specified for any type and hence is useless, would only fire for not-well typed data.

Two anomaly heuristics have been developed based on these notions. The first is based on the notion of 'type-unknown' rules and generates a warning and an explanation for rules that are 'type-unknown'. The second heuristics generates an error for rules that are type incorrect. Both heuristics are realized as rules that reason about the rules in the knowledge base (the details of this are described in detail in the section 6.4 below).

Anomaly Heuristic 3 (Method Undefined for Variable in Rule) *A warning is generated for all rules that use a method on a variable that is not known to be of a type that defines this method. A variable A is said to be known to be of a type b in a rule, iff the rule also contains a positive atom of the form A:c, where c::b.*

This heuristic is able to alert the developer to problems like rule 2 in the example above. A second heuristic identifies some of the statically type-incorrect rules.

Anomaly Heuristic 4 (Rule Method Unknown) *An error is generated for each rule that uses a method that is not defined in any signature atom.*

This second heuristic can statically detect some type incorrect rules, such as rule 3 in the example above. This heuristic was generated mostly to ensure that developers are reminded to update rules after changes to the signatures.

6.3 Bug Pattern based Anomalies for F-logic

Bug patterns, i.e. patterns in the rule base that are often the consequence of a fault, are used in addition to the type checking based anomalies described above. These bug pattern based anomalies can be coarsely grouped into two classes: those mainly focusing on rules and those focusing on the class hierarchy; each of these is detailed in one subsection below.

6.3.1 Rule Bug Patterns

Three heuristics are defined that are mainly concerned with rules, identifying the use of non-existing instances in rules, unsafe rules and undefined classes.

Anomaly Heuristic 5 (Rule Body Instance Undefined) *A warning is generated for each rule that uses an instance in the rule body that is not defined elsewhere.*

Consider the following example rule base:

```
FORALL A
  A:VW_car
<-
  A[maker->VW]
```

A warning[5] of the type 'Rule Body Instance Undefined' would be generated for this rule base, because `VW` is an instance that is not defined in the rule base. This kind of problem would most likely be caused by some change that removed or renamed the VW entity that was present at some point.

Anomaly Heuristic 6 (Head Variable Unbound) *An error is created for each rule that uses a variable in the head, but not in at least one positive atom in the body.*

Rules that fit this pattern are usually not intended in this way, and, more importantly, the Ontobroker inference engine cannot process such rules and will terminate the reasoning process with an exception and without computing a result.

Anomaly Heuristic 7 (Class Undefined) *A warning is created for each class that is used in an is-a statement but that is not defined elsewhere. For this heuristic the is-a statements in rules and in data atoms are considered.*

Again this heuristic is mostly geared towards identifying faults caused by the independent evolution of rules and the class hierarchy; i.e. to detect cases where a rule wasn't changed after a class had been removed or renamed.

[5]In fact, some type checking based errors defined in the previous section would be raised as well.

6.3.2 Class Hierarchy Bug Patterns

Four heuristics are defined that deal mainly with the class hierarchy and support the developer in maintaining it.

Anomaly Heuristic 8 (Many Subclasses) *A warning is created for each class with more than 12 subclasses. Such a large number of subclasses is often seen as bad practice [120] and results in hard to maintain ontologies.*

Anomaly Heuristic 9 (Few Subclasses) *A warning is created for each class with less than two subconcepts. A very small number of subclasses is also often seen as bad practice [120] and indicates possibly superfluous classes.*

Another indicator of possible superfluous classes are those that do appear in the class hierarchy but that aren't used as types for instances or in conditions for rules.

Anomaly Heuristic 10 (Class Leaf) *A warning is created for each class that has neither instances, nor subconcepts nor is used in the atom of a rule.*

While such classes might have been created on purpose (e.g. classes that are intended to be instantiated within the context of tests) they often also indicate loose ends in the knowledge base; where the developer created a class for some purpose that was not realized.

Anomaly Heuristic 11 (Root Classes) *An information is generated for each concept that has no superconcepts (other than itself).*

Finally a list of root classes is created and presented to the user. Obviously being a root concept can be intended and, in fact, every class hierarchy needs at least one root concept. However, a root class may also indicate a class that the developer has forgotten to sort into the class hierarchy or that is left over from a restructuring of the class hierarchy. Hence, and because the number of root concepts is usually relatively small, this list of root concepts is presented to the user.

6.4 Implementation

All anomaly heuristics described in the previous section are realized with meta-reasoning in F-logic. Each anomaly heuristic consists of a number of

F-logic rules that are described in an XML file. This XML file also contains information about parameters a heuristic accepts and natural language patterns that enable the user-level explanation of the anomalies identified. An user interface integrated into DarkMatterStudio allows to parameterize the rules and to look at the explanations.

In the current section the implementation of the anomaly heuristics is described in detail. The section starts with a description of rule reification; the data structures that enable the system to reason about the rules in its rule base. Next, the realization of the actual heuristics is described before the user interface is introduced.

6.4.1 Rule Reification

Reification is the process of turning an abstraction into an object that can be reasoned about. In the context of the work presented here, rules are reified so that they can be examined and reasoned about with anomaly heuristics.

Reification of F-logic rules was a pre-existing feature of the Ontobroker inference engine [59] and is introduced here using an example; the complete ontology for the representation of the reified rules is shown in appendix D.

As an example consider the following rule that ensures the reflexivity for a method m, i.e. that, if calling the method m on an instance B returns A, calling the same method on A returns B.

```
RULE reflex:
FORALL A,B
  A[m->B]
<-
  B[m->A]
```

Please recall that the F-logic rules are translated to normal logic before they are processed by the inference engine (see section 2.2.3); the reification is also applied to these translated rules. For the example rule above, the normal logic version is the following:

$$att(A, m, B, DefaultModule) \leftarrow att(B, m, A, DefaultModule)$$

Based on on this translated version, the facts created in the reification are the following; the reified rule is:

```
reflex:Rule[
  identifier->reflex;
  bodyVariables ->> {variable(A),variable(B)},
  headVariables ->> {variable(A),variable(B)},
  heads -> [literal1];
  bodies -> [literal1];
  bodyLength -> 1.0;
  headLength -> 1.0;
  module -> DefaultModule].

literal1:Literal[
  atom->atom3;
  variables ->> { variable(A),variable(B) }
  isPositive -> true;
  module -> DefaultModule].

atom3:PAtom[
  predicateSymbol -> "att";
  arguments ->[variable(A), constant(m),
    variable(B),constant(DefaultModule)];
  arity -> 4.0].

literal2:Literal[
  atom->atom4;
  variables ->> {variable(A),variable(B)}
  isPositive -> true;
  module -> DefaultModule].

atom5:PAtom[
  predicateSymbol -> "att";
  arguments ->[variable(B), constant(m),
    variable(A),constant(DefaultModule)];
  arity -> 4.0].
```

The top level instance representing the reification of a rule is `reflex`, an instance of the class `Rule`. This instance has methods with the variables used in head and body, references to instances representing the literals in head and body[6], the numbers of atoms in head and body and finally the module of the rule.

The literal instances each have one atom, their module, the variables used in them and, most importantly, a method indicating whether they are negated or not.

Finally the atom instances have methods for the predicate symbol, its arity and a list of the arguments.

[6]Note that there are only atoms, not literals in the rule head. Having literal instances represent the rule head is a quirk in the reification ontology.

Since, as defined in section 2.1, normal logic rules consist of only a head atom and body literals, all of these can be represented in the presented way. Please note, however, that reifying the normal logic rules (and not the F-logic rules directly) means that one F-logic rule can be represented by any number of reified rules. The link between the reified rules and the F-logic rules they where derived from is given by the `identifier` method of the rule instance.

With this reification ontology it is now possible to reason about all elements in the rule base, the facts being already accessible with the standard elements of F-logic.

Note that the reification here is only a representation of the rules for introspection. It is not, however, possible to use this ontology to create new rules at runtime; it is not possible, that rules infer new rules.

6.4.2 Anomaly Heuristic Realization

Anomaly heuristics are rules that jointly define a special 4-ary predicate:

`problem(anomalyId,severity,module,location)`

The arguments for this predicate are:

- **anomalyId** The kind of anomaly that was found.
- **severity** 'Error', 'Warning' or 'Information'; indicating the severity of each anomaly.
- **module** The module the anomaly was found in[7].
- **location** A list of further instances giving more information about the location and the kind of anomaly. The first element of the list is used to group anomalies, all further elements are used only by the explanation templates of the heuristics (see below).

An example anomaly is given below, it shows the anomaly identified by the anomaly heuristic 'Rule Method Unknown' when applied to the reflex rule introduced above. Please note that the actual anomaly heuristics use a terminology different from that in this thesis (e.g. 'property' instead of 'method'), this is due to non-standard terminology being used in Project Halo in an attempt to make F-logic easier to understand.

```
problem(rule_property_unknown,severity_error,
   DefaultModule,[reflex,m])
```

[7]Having the module as one argument of the problem predicate allows - together with the datalog style reasoning in Ontobroker - to compute anomalies only for certain modules.

As arguments the problem predicate has the id of the anomaly heuristic (`rule_property_unknown`), an instance indicating that this is classified as an error (`severity_error`), the module (`DefaultModule`) and a list with further information about the details of the anomaly - in this case the identifier of the rule (`reflex`) and that of the method (`m`).

The anomaly heuristics themselves are defined in an XML file, allowing for easy extension, change and exchange without needing any change in the code of the ontology engineering environment or the inference engine. An example for the anomaly heuristic 'Rule Method Unknown' is shown below. All elements appearing in the example are detailed therafter.

```
<title>Rule Property Unknown</title>
<id>rule_property_unknown</id>
<category>Rule</category>
<performanceIndex>1</performanceIndex>
<defaultOn>true</defaultOn>

<code>
  RULE rule_property_unknown:
    FORALL R,P,M,E
    problem(rule_property_unknown,severity_error,M,[R,P])
  <-
    is_rule(R,M) AND
    att_rule(R,E,P) AND
    NOT property(P,M).
</code>

<description> A rules uses an attribute or relation that
  is not defined anywhere. Run this test after you
  removed attributes or relations from the ontology that
  some rules may still use.
</description>

<explanation>The rule $0$ uses the property $1$ - such
  a property is not defined for any concept! This rule
  will not work.
</explanation>
```

The definition of an anomaly heuristic consists of the following parts:

- **Title** The name of the anomaly heuristic. This is shown in the user interface.
- **ID** The id of the anomalies created by this anomaly heuristic; this is used to find the explanation template for a given anomaly. This id needs to be unique.
- **Category** A category of the anomaly heuristic; used to group anomalies in the user interface.

- **Performance Index** A number indicating how long a particular heuristic usually takes to execute. This is used in the user interface as a guide for the user.
- **Default On** Indicating whether the anomaly heuristic should be run per default, when the user makes no configurations.
- **Code** The actual rule(s) realizing the anomaly heuristic. The anomaly heuristic rules can use a library of helper rules for meta-reasoning about F-logic programs. This library too was created as part of this thesis and is shown in appendix E.1. The example above uses the predicates `is_rule` (is_rule(Rule,Module) - all rules in one module), `att_rule` (att_rule(Rule, Entity, Property) - all methods/properties used in a rule on an entity/instance) and `property` (property(Property,Module) - all instances defined as properties) that are all defined by this library.
- **Description** A description of the anomaly heuristic that is displayed to the user as further information.
- **Explanation** A template that is used to explain anomalies found by this heuristic to the user. The explanation template contains numbers enclosed in dollar signs, these are replaced with the instances from the list in the problem predicate. Applying the explanation template to the example anomaly above yields: *The rule reflex uses the property m - such a property is not defined for any concept! This rule will not work.*

Not shown in this simple example is the feature that allows users to parameterize an anomaly heuristic. For example the 'Many Subclasses' anomaly heuristic introduced above allows to change the threshold over which the number of subclasses gets reported. The realization of this feature is shown below in a excerpt from the definition of the 'Many Subclasses' heuristic:

```
<code>
FORALL Concept_,NumberSubclasses_, Module
  problem(style_manySubclasses,severity_warning,Module,
    [Concept_,NumberSubclasses_])
<-
  number_subclasses(Concept_,NumberSubclasses_,Module) AND
  greater(NumberSubclasses_,$num$).
</code>

<parameters>
  <parameter>
    <name>Number Subclasses</name>
    <id>num</id>
    <defaultValue>12</defaultValue>
    <description>The number of subclasses that are allowed
```

```
        before a warning gets created.
      </description>
    </parameter>
  </parameters>
```

Here the rule realizing the anomaly heuristic includes a parameter `$num$` that is replaced by a user supplied value before the anomaly heuristic is executed. The description of the parameter, its default value and how it is explained to the user is done with further XML elements.

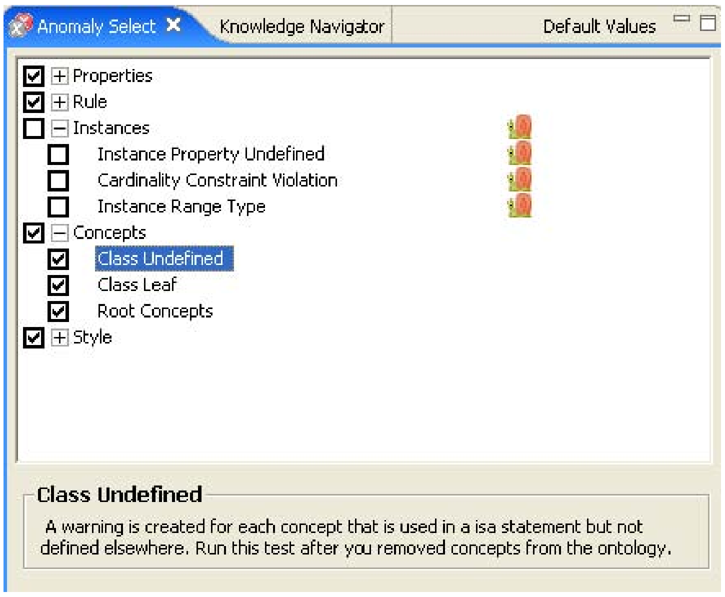
The actual realization and execution of the anomaly heuristics comprises of the following steps:

- On system startup the XML file specifying the anomaly heuristics is read, validated and the user interface is configured such that the user is able to choose and parameterize the anomaly heuristics. A default configuration of the anomaly heuristics is created based on the configuration from the last time the system was run or (if this is not available) from the default values.
- Any time the system runs, the user can select, deselect or parameterize anomaly heuristics.
- At any time the user can choose to run the anomaly heuristics based on the current configuration; this is done in the following steps:
  1. The anomaly heuristic parameters are inserted into the rules.
  2. The rules from the currently active anomaly heuristics and the meta-reasoning helper rules are added to the rule base.
  3. A query for all problems in the user modules is executed. The result is saved.
  4. The anomaly rules and the helper rules are removed from the rule base (to not hinder performance or confuse the user).
  5. The explanation templates are used to generate natural language texts explaining the anomalies found.
  6. The anomalies are shown to the user.

6.4.3 User Interface

The interface for the anomaly component consists of two views that are integrated into the Dark Matter Studio system. The first view shows information about the available heuristics, gives hints on when to enable them and allows their configuration. The other view shows the actual anomalies found in the rule base.

The *anomaly select view* (see figure 6.1) shows the different anomalies and whether they are currently enabled, i.e. will run the next time DMS searches for anomalies. The different anomaly heuristics are ordered in broad categories such as rules, instances or concepts. The anomaly select view also allows to activate, deactivate and configure anomalies.



- [✓] Properties
- [✓] Rule
- [ ] Instances
  - [ ] Instance Property Undefined
  - [ ] Cardinality Constraint Violation
  - [ ] Instance Range Type
- [✓] Concepts
  - [✓] Class Undefined
  - [✓] Class Leaf
  - [✓] Root Concepts
- [✓] Style

Class Undefined

A warning is created for each concept that is used in a isa statement but not defined elsewhere. Run this test after you removed concepts from the ontology.

Figure 6.1: *Anomaly select view.*

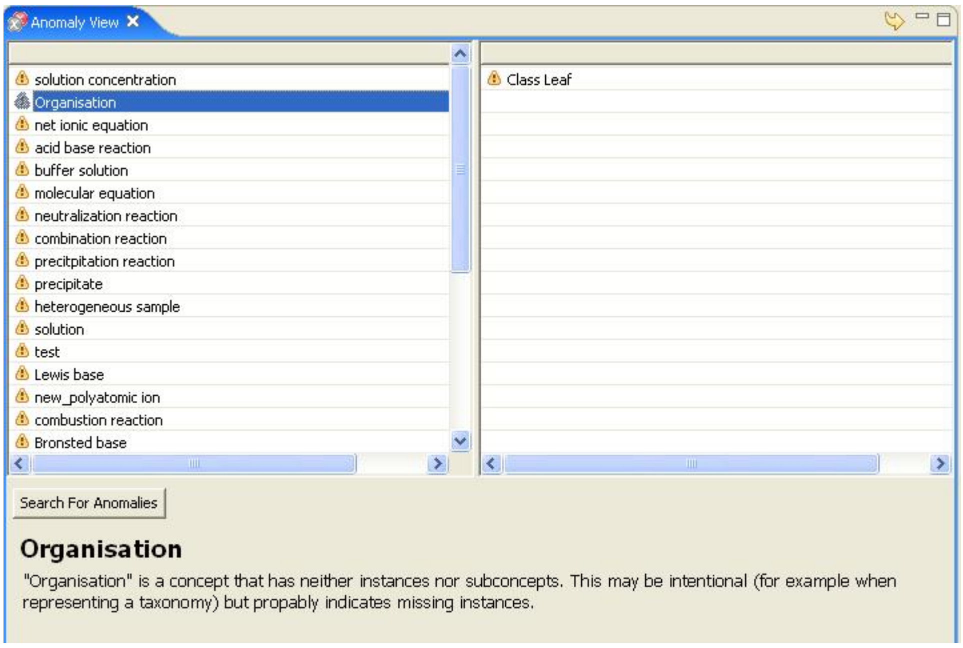
The second view; the *anomaly view* displays the anomalies that have been found in the rule base (see figure 6.2). The anomalies are grouped by either the affected entity or the type of anomaly.

Clicking on an entity shows the different anomalies affecting this entity. All anomalies that have been found are displayed with a small icon indicating their severity (in the example only warning - indicated by a yellow warning sign - are shown). A short explanation text is displayed below the list of anomalies.

6.5 Prior Work

Anomaly heuristics to support the creation of rule based systems have been researched and developed since the days of the first expert systems.

Probably the first anomaly detection system was included in the Teiresias system [45]. This system examined the rule base and created a model of the patterns in which attributes were used together. On newly introduced rules



solution concentration
Organisation
net ionic equation
acid base reaction
buffer solution
molecular equation
neutralization reaction
combination reaction
precitpitation reaction
precipitate
heterogeneous sample
solution
test
Lewis base
new_polyatomic ion
combustion reaction
Bronsted base

Class Leaf

Organisation

"Organisation" is a concept that has neither instances nor subconcepts. This may be intentional (for example when representing a taxonomy) but propably indicates missing instances.

Figure 6.2: *Anomaly view.*

the system would then create a warning when this pattern was violated, e.g. 'all previous rule considering age and glucose level also considered the weight of the patient'.

From this grew a first generation of anomaly detection systems [133]. This first generation focused on the detection of anomalies involving only one or a very small number of rules. Typically they could detect the following problems:

- **Duplicate rules:** Pairs of redundant rules, where one rule duplicated the conditions and conclusions of the other. One rule could also be subsumed by the other.
- **Missing rules** Missing rules were detected based on possible combinations of input values that do not match any rule body.
- **Contradicting rules** Pairs of rules that, on the same situation, conclude incompatible facts.

Typical system from this first generation of anomaly detection systems are RCP [158] and CHECK [119].

The second generation of anomaly detection systems focused on identifying redundancies and contradictions across large groups of rules. These systems also attempted to find more sophisticated cases of missing knowledge. Important systems are EVA [33], COVER [134], and KB-REDUCER [68].

Based on experiences from these systems, Preece and Shinghal [133] created a very influential categorization of anomaly heuristics. They differentiated the following groups:

- Redundancy, further divided into:
  - Unfireable rule
  - Subsumed rule
  - Unusable consequent
- Ambivalence, in particular contradicting rules
- Circularity
- Deficiency, in particular unused inputs.

Only the first of these, redundancy is examined in the context of this work. Circularity is not a problem with normal logic programs[8]. Deficiency as unused input is not detectable, since the input is not known or specified in advance. The system presented here, however, extends this classification in some aspects, since it also considers some anomalies related to the ontology.

Most of the work on anomaly detection for rule based systems has been done more than one decade ago, very few novel work exists. Notable recent works are attempts to extend the anomaly idea beyond rules to specification artifacts [165] and attempts to use binary decision diagrams to speed up the detection of anomalies involving large numbers of rules [114]. A comprehensive approach is the VALENS (VALid Engineering Support) tool for validation and verification of Aion knowledge bases. This system realizes anomalies from all groups identified by Preece and Shinghal.

The system presented in this chapter is the first attempt at creating an anomaly detection system for F-logic. Its also the first system to realize some static type checking for F-logic. It does not, however, pioneer entirely new classes of anomalies or novel ways to compute them; rather it focuses on the practical aspects of embedding them in a flexible, transparent and extensible way in a modern rule base engineering environment.

[8]Unless paired unfortunately with negation, causing a rule base to become unstratisfiable.

6.6 Conclusion

This chapter presented an anomaly detection framework for F-logic. It supports anomaly heuristics based on the F-logic type system, problems in rules and problems with the ontology. The type based anomalies realize both the yardstick type semantics of F-logic and heuristic static type checking.

The framework is implemented as a plugin for the rule engineering environment Dark Matter Studio and (except for the user interface) it is implemented in F-logic rules processing a reified version of the rule base. The created framework is easily extensible and relatively portable to other systems due to all anomaly heuristics being defined in XML files and realized in F-logic itself. The implementation supports template based natural language explanation of the problems found; the templates too, are contained in the XML files.

The presented framework is the first anomaly detection system for F-logic and includes the first implementation of static type checking for F-logic. Compared to the related work in the broader area of anomaly detection for any rule based system it is unique in its XML based extensibility.

The most important limitation of the proposed approach is the current absence of a systematic evaluation. Obtaining existing rule base and using the presented system to identify faults in these is hence the most promising and most important venue for future work. Another possible venue for future work is the extension of the current set of anomaly heuristics.

Chapter 7

Visualization

The analysis in chapter 3 identified the action-mistake transition in the fault lifecycle (the proportion of developer actions that are mistakes) as one of the critical points for explaining the debugging challenge of rule base development. The identification of this transition as problematic rested on three observations (explained fully in section 3.2):

- **The Problem of Terminology**: that while rules are often seen and presented as self-contained entities, they depend on a consistent domain formalization across all rules they interact with.
- **The Problem of Opacity**: that the behavior of a rule base is determined by the interaction of the rules; that at the same time, however, this interaction is usually not shown to the user.
- **The Rise of End User Programmers**: that less trained developers are creating rule bases.

To tackle this challenge a visualization of the rule base was conceived. This visualization shows the structure of the rule base on the basis of the actual rule interactions. It should enable the developer to make more correct decisions by making her more aware of the connections in the rule base; i.e. with which parts of the rule base her changes need to be consistent with and which other rules may be affected by a change. This visualization is an implementation of the **visibility** design principle required above (see section 3.3.2), it further also realizes the **declarativity** principle by being completely independent of the evaluation strategy of the inference engine.

The visualization of entire rule bases, independent of the answer to any particular query, is a problem that has so far been largely ignored by the research community. The goals for such a visualization are the same as for UML class diagrams and other overview representations of programs: to aid teaching, programming, debugging, validation and maintenance by facilitating a better understanding of a computer program by the developer. The importance of such representations cannot easily be overstated and is probably higher than in object oriented programming: In object oriented programming the overall structure is explicitly created by the programmer with links between objects, the inheritance hierarchy and method calls; in rule bases the interaction between rules is only decided by the inference engine and usually never shown to the user. Hence an overview representation of the entire rule bases is necessary to make this hidden structure visible. Before suitable visualizations for rule bases can be created, however, we need a definition of the *overall structure of the rule base* - doing this with respect to usage data is the main contribution of this chapter.

This chapter starts with a recapitulation of the static structure of the rule base. Next, the dynamic structure of a rule base is introduced in section 7.2. Section 7.3 introduces techniques to hide[1] and to join multiple rules[2] in the visualization. The implementation of this visualization is described in section 7.4, before an overview of related work and the conclusion.

It is important to note that large parts of the actual implementation presented in this chapter were implemented by Imen Borgi, a student, with supervision by the author.

7.1 Static Structure

In this section the static structure of the rule base is quickly recapitulated and presented from the viewpoint of visualizations; a definition is already given in section 5.1.2 of the debugging chapter.

The core concept for the static structure of rule bases is the *depends-on* connection between rules. Such a link indicates that two rules seem to be able to work together. Consider the following rule base:

$$father(A) \leftarrow male(A), parent(A, B) \tag{1}$$

$$parent(X, Y) \leftarrow child(Y, X) \tag{2}$$

$$employer(X) \leftarrow owns(X, C), employed\_at(A, C) \tag{3}$$

It can be seen that, with the right facts, rule (1) could work with the result of rule (2). The following fact base consisting of only two facts should serve as an example:

$$male(mike), child(michelle, mike)$$

[1]For example the F-logic axioms.

[2]For example all rules in a module or multiple rules created from the compilation of one high-level rule.

With these facts, rule (2) could deduce that $parent(mike, michelle)$ from $child(michelle, mike)$ and rule (1) could then deduce that $father(mike)$. In such a case it is said that rule (1) *depends-on* rule (2). It is said depends-on, because the rule (1) could not have made the inference that Mike is a father without rule (2) being present; for this conclusion it depended on the other rule. For the depends-on connection, only those cases are considered, where a rule works directly on the conclusion of another rule; not the cases where there are intermediary rules. The depends-on connections are calculated on the rules without consideration for the actual facts that are available: a depends-on connection exists independently of the facts. A depends-on connection between a rule A and a rule B states that there exists a set of facts such that an inference of rule A directly depends on rule B being present.

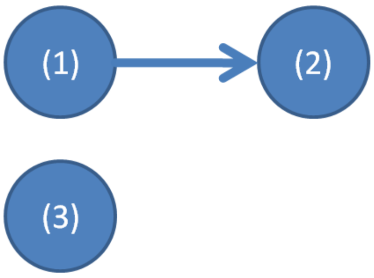


Figure 7.1: *Static structure of the example rule base in section 7.1.*

The rule (3) on the other hand, has no depends-on connection to any of the other rules - there exists no set of facts such that rule 3 could directly work on data inferred by one of the other two rules. Also note that depends-on connections are not symmetric: rule (1) depends-on (2) but not the other way around.

A simple way to calculate an approximation of the depends-on connection is the following: a depends-on connection exists from rule A to rule B, iff rule A has a body atom that can be unified with the head atom of B. This way to calculate the depends-on connections is only an approximation because it only considers single atoms and not the entire rule; other atoms in the rule body may make a depends-on connection impossible. A definition of the static structure as well as links to other ways to compute these connections is given in section 5.1.2.

The rule graph for the simple rule base example in this section is shown in figure 7.1. It quickly conveys information such that {1,2} and {3} are independent parts of the rule base, that a deletion of (2) will affect (1) and that someone interested in understanding rule (1) should also look at (2). For a rule base consisting only of three rules such a diagram is obviously not needed, but it can be seen how this kind of information can aid the creation, debugging, maintenance and reuse of larger rule bases.

7.2 Dynamic Structure

The static structure defined in the previous section represents the potential of rules to interact, without making assumptions/using information about the queries and facts a rule base is used with. The dynamic structure of a rule base complements this by combining the prooftrees from all known queries to the rule base into an overview picture. The resulting picture shows which rules and rule connections actually matter in the use of a rule base. When used to visualize the prooftrees from the tests for a rule base it also gives a picture of test coverage.

Consider the following rule base. This rule base makes heavy use of a type hierarchy encoded in special predicates - a structure typical for rule bases created in F-logic [92].

$$is\_a(A, working\_parent) \leftarrow is\_a(A, parent), is\_a(A, employee) \tag{4}$$

$$is\_a(A, mother) \leftarrow is\_a(A, female), child(X, A) \tag{5}$$

$$is\_a(A, C) \leftarrow is\_a(A, B), subclass(B, C) \tag{6}$$

The static structure of this rule base is confusing (see figure 7.3) as almost everything depends on everything else; looking at the rule interactions in the actual use of the rule base can help here.

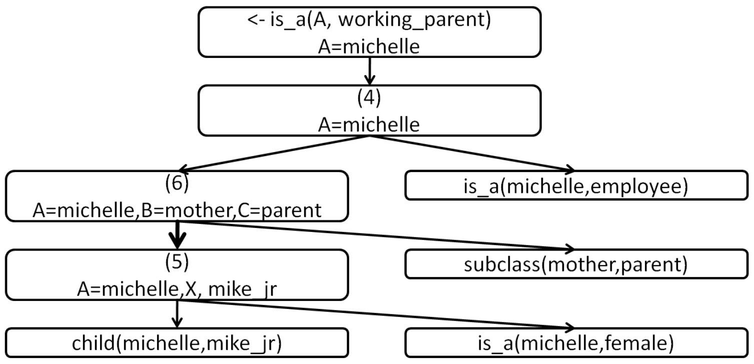


Figure 7.2: *An example prooftree*

The structure that represents the interaction of rules and facts in the inferring of a result is called a prooftree. The root node of a prooftree is always the query, its variables bound to the result that has been returned. The children of this node are the rules that were directly needed to prove the query.

The children of these rules are again the rules needed to prove them. Leafs of the prooftree are formed by the facts in the knowledge base. A formal definition of prooftree has been already given in section 4.2.1.

An example prooftree is created by the query $is\_a(A, working\_parent)$ posed against the rule base above and the following facts:

$$subclass(mother, parent), is\_a(michelle, female)$$
$$is\_a(michelle, employee), child(michelle, mike\_jr)$$

A graphical representation of this prooftree is shown in figure 7.2. It shows the query on top with the variable A bound to michelle. This result of the query directly depended on the firing of rule (4) that in turn depended on rule (6) and the fact $is\_a(michelle, employee)$ etc.

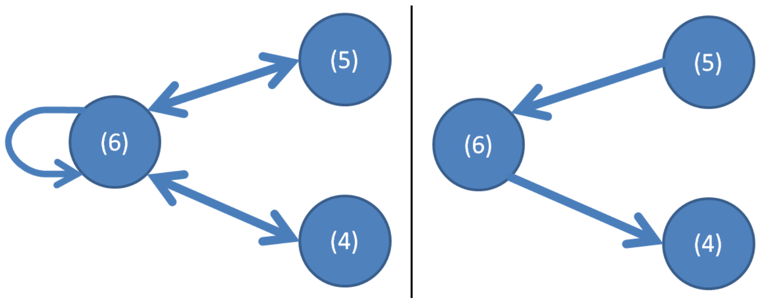


Figure 7.3: *The static structure (to the left) and the dynamic structure (right) of the example in section 7.2. In this example the dynamic structure is very similar to a prooftree - because the usage data consits only of this one prooftree; this is not the case generally.*

The novel approach presented here now combines all known prooftrees for a rule base into an aggregated picture of the dynamic structure of the rule base, independent from the answer to any particular query. To do this, first usage data is defined as the multiset of all known prooftrees for a rule base, i.e.

Definition 30 (Rule Base Usage Data) *Rule base usage data* $U = \{G_{P,1}, G_{P,2}..., G_{P,i}\}$ *is the multiset of all known prooftrees for a rule base.*

The prooftrees could be created by tests of the rule base or be collected during the actual use of the system. Based on this usage data, the dynamic structure of the rule base can be defined. The dynamic structure is based on the static structure defined above, but adds a weight for each arc based on how often an arc has been exercised in the usage data.

Definition 31 (Dynamic Structure of a Rule Base) *The dynamic structure of a rule base R, is a directed graph $G_R = (R, A)$ and a function $f_s : A \rightarrow \mathbb{N}$ with*

- *The set of vertices R, there is one vertex for each rule in the rule base. We write that R_{E_1} is the vertex for rule E_1.*
- *The set of ordered pairs A of vertices called arcs or arrows. An arc $e = (R_{E_1}, R_{E_2}) \in A$ exists iff at least one body atom of E_1 can be unified with at least one head atom of E_2.*

The value of the function f_s for an edge $e = (R_{E_1}, R_{E_2})$, $f_s(e)$ is the number of all edges (p_{r1}, p_{r1}) in the prooftrees of the rule base usage data where $f_p(p_{r1}) = R_{E_1}$ and $f_p(p_{r2}) = R_{E_2}$.

Intuitively, the dynamic structure can be understood as the combination of all prooftree known for a rule base. The weight of an arc from rule a to b can then be understood as the number of times that rule a depended on rule b in the actual use of a rule base. Graphical representations of this structure can use this weight to hide unused depends-on links and to highlight important rules and links[3].

The dynamic structure of the example in this section is shown in figure 7.3, an example for a larger rule base is shown in section 7.3.3. Compared to the static structure it gives a much better picture of the interactions between these rules as they would happen for real world data. The dynamic structure facilitates a better understanding of the rule base for programming, maintenance, test coverage, reuse and even profiling.

Displaying the dynamic structure in this way is not the only possible way to use the usage data for a rule base. The following section will discuss how it can also aid transformations of the rule graph to hide certain rules or to display the rule graph at different levels of abstraction.

7.3 Hiding and Joining Rules

The techniques presented so far work for showing the structure of a toy rule base. Even small real life rule bases, however, pose additional challenges: they are too large to show every rule, very general rules/axiom like rules (similar to rule (6) in the previous section) confuse the picture, and sometimes graphical editors are used that create multiple rules for one high level entity known to the user. For instance, figure 7.7 shows the static structure

[3]In the implementation that was used to create the screenshots for this chapter, the weight of arcs was only used to hide arcs with a weight of 0.

of a small but functional rule base - the structure is much too confusing to be of any use.

To deal with these problems, algorithms are presented to hide and join some rule nodes, while still keeping the overall structure of rule interactions intact.

7.3.1 Hiding Rules

For certain very general, axiom like rules it makes sense to completely hide them from the view of the user. This is particulary important for F-logic rule bases that add a number of axioms to the rule base that are transparent to the user[4]. Again the usage data is used to ensure that only relevant links are displayed. Consider the following example rule base, slightly extended from the rule base in section 7.2.

$$is\_a(A, working\_parent) \leftarrow is\_a(A, parent), is\_a(A, employee) \tag{7}$$
$$is\_a(A, mother) \leftarrow is\_a(A, female), child(X, A) \tag{8}$$
$$is\_a(A, C) \leftarrow is\_a(A, B), subclass(B, C) \tag{9}$$
$$is\_a(A, laptop) \leftarrow portable(A), is\_a(A, computer) \tag{10}$$

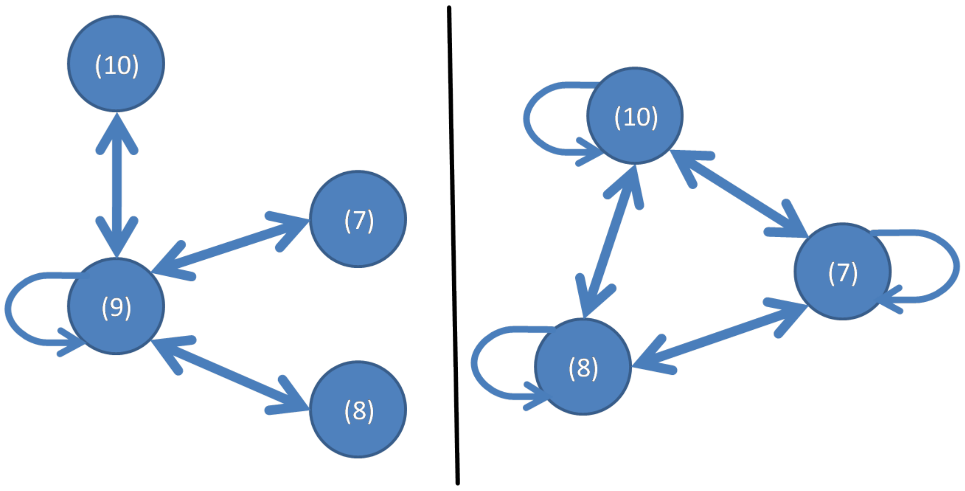


Figure 7.4: *The static dependencies for the example in section 7.3.1. The left side shows the unaltered dependencies, the right side the dependencies after rule (9) has been trivially hidden.*

[4]This is described in detail in section 2.2.6.

For this example, we assume that the very general rule (9) should be hidden while preserving the interaction structure of the rule base as much as possible. The left side of figure 7.4 shows the static dependencies between the rules with rule (9) still being shown. A naive algorithm to hide one rule would connect all vertices that depend on the rule node being hidden to all vertices the hidden rule depended on (see algorithm 4). This algorithm, however, normally adds a large number of depends-on connections and thereby makes the visualization less comprehensible. The right side of figure 7.4 shows the example rule base after rule (9) has been hidden using this naive algorithm. The visualization of the transformed example shows that every rule depends on all rules (including itself) - resulting in a visualization without much useful information.

```
Data: A directed graph $G_R = (R, A)$, representing the static structure of
      the rule base, and a rule $r_1 \in R$ that is to be hidden.
Result: A transformed graph $G'_R = (R', A')$, representing the static
        structure of the rule base while hiding $f$.
foreach $a_i = (r_i, r_j) \in A$ do
    if $r_i = f$ then
        Add $r_j$ to the set $Outgoing$;
        Remove $a_i$ from $A$;
    end
    if $r_j = f$ then
        Add $r_i$ to the set $Incoming$;
        Remove $a_i$ from $A$;
    end
end
foreach $r_i \in Incoming$ do
    foreach $r_o \in Outgoing$ do
        Add $a_{io} = (r_i, r_o)$ to A;
    end
end
Remove $f$ from R;
```

Algorithm 4: *A naive algorithm to hide rules.*

Usage data in the form of prooftrees can help to better deal with this problem, because it contains *paths* through the rule base, not only arcs between two rules. When hiding rules this enables to retain only the actual paths through the hidden node(s). A detailed description of this is shown as algorithm 5. The algorithm presented works by removing the rule that is to be hidden from all prooftrees and then re-creating a rule graph from the transformed prooftrees. For readability the algorithm shown deals only with one rule that is hidden, however, it can be trivially extended to hide multiple rules.

Figure 7.5 shows the visualization of the example rule base presented at the beginning of this paper after rule (9) has been hidden with this algorithm. The usage data used for this computation consisted of the prooftrees resulting from the queries $is\_a(A, working\_parent)$ and $is\_a(A, portable\_computer)$ being posed against the rule base and the following facts:

$$subclass(mother, parent), is\_a(michelle, female)$$
$$is\_a(michelle, employee), child(michelle, mike\_jr)$$
$$subclass(mac, computer), is\_a(myPowerBook, mac)$$
$$portable(myPowerBook)$$

Compared to the right side of figure 7.4 this example visualization is a lot simpler. It gives a clear picture of the rule interactions - possibly even a clearer picture than before rule (9) was removed.

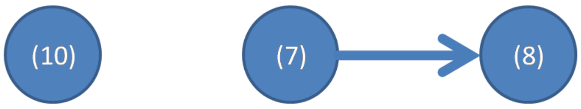


Figure 7.5: *The dependencies for the example in section 7.3.1 after rule (9) has been hidden with the algorithm that utilizes usage data.*

7.3.2 Joining Rules

There are a number of cases where it makes sense to join a number of rules for display purposes:

- Some rule engineering environments create more than one rule for a high level-entity edited by the user[5]. Here the visualization should only show the high-level entity without losing information.
- In a large rule base there is often some hierarchical structure on top of the rules (like rule packages or different files defining rules). The system can join all rules in each package and thereby give a high-level view of the rule base, allowing to *zoom in* at one package, to show the rules in a package.
- In large rule bases without auxiliary structure, clustering algorithms could be used to identify and jointly display rule clusters, again allowing to expand the clusters.

[5]For example the DarkMatterStudio system described in section 3.1.2.

```
Data: The multiset of all known prooftrees for a rule base
      $U = \{G_{P,1}, G_{P,2}..., G_{P,i}\}$, a rule $r_h$ that is to be hidden and the
      function $f_r$ that returns the rule for a prooftree node.
Result: A rule graph $G_R = (R, A)$, representing the static structure of the
        rule base while the rule $r_h$.
foreach Prooftree $G_{P,i} = (V_i, A_i) \in U$ do
    foreach $a_x = (v_u, v_v) \in A_i$ do
        if $f_r(v_u) = r_h$ then
            Add $v_v$ to the set $Outgoing$;
            Remove $a_x$ from $A$;
        end
        if $f_r(v_v) = r_h$ then
            Add $v_u$ to the set $Incoming$;
            Remove $a_x$ from $A$;
        end
    end
    foreach $r_n \in Incoming$ do
        foreach $r_o \in Outgoing$ do
            Add $a_{no} = (r_n, r_o)$ to $A_i$;
        end
    end
    foreach $a_x = (v_u, v_v) \in A_i$ do
        Add $f_r(v_u)$ to R;
        Add $f_r(v_v)$ to R;
        Add $a_y = (f_r(v_u), f_r(v_v))$ to $A$;
    end
end
```

Algorithm 5: *An algorithm to hide rules while using usage data.*

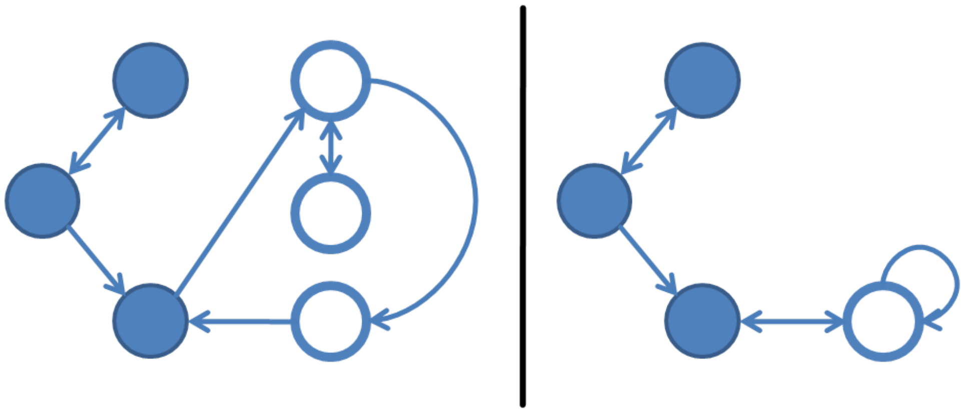


Figure 7.6: *An example showing the joining of rules. The left side shows the rule base before, the right side after the rules represented by white circles have been joined.*

Joining of rules is done by replacing a number of nodes in the rule graph with one new node. The arcs that go to or from one of the nodes being joined get changed to end/start at the newly introduced node. The rule connections used as basis can be either the static structure of the rule base or the structure representing the usage data. The detailed algorithm for this transformation is shown as algorithm 6. An example of this algorithm applied to a small rule base is shown in figure 7.6.

7.3.3 Example

The right side of figure 7.7 shows the same rule base as the left side after usage data has been used to scale the rules nodes, axioms have been hidden and some rule sets (reflecting rules automatically created from one high level entity) have been joined. These transformations have uncovered the relatively simple, layered and ordered structure of this diagnostic rule base. The rule base used for this example consists of 15 user created rules high level that got translated into 54 F-logic rules. The usage data for this example consists of 15 test queries resulting in 26 prooftrees. There are more prooftrees than queries because some queries return more than one result.

The size of the rule base that can be visualized with the approach here is not restricted to such small rule bases. Because of the algorithm to join groups of rules, the same algorithm can be used to show the interaction between rule modules, or sets of rules. This approach also allows to look at one module in detail while showing the rest of the rule base joined into modules.

```
**Data**: A directed graph $G_R = (R, A)$, representing the structure of the rule
        base, and a set of rules $S = \{r_{j1}, r_{j2}, ..., r_{jn}\}$ that is to be joined.
**Result**: A transformed graph $G_R^{'} = (R, A)$, representing the structure of
        the rule base with the joined rules.
foreach $a_x = (r_i, r_j) \in A$ do
    if $r_i \in S$ then
        Add $r_j$ to the set $Outgoing$;
        Remove $a_x$ from $A$;
    end
    if $r_j \in S$ then
        Add $r_i$ to the set $Incoming$;
        Remove $a_x$ from $A$;
    end
end
Remove all $r \in S$ from R;
Add a new rule $r_j$ to R;
foreach $r_i \in Incoming$ do Add $a_x = (r_i, r_j)$ to A;
foreach $r_o \in Outgoing$ do Add $a_x = (r_j, r_o)$ to A;
```

Algorithm 6: *Algorithm to join rules.*

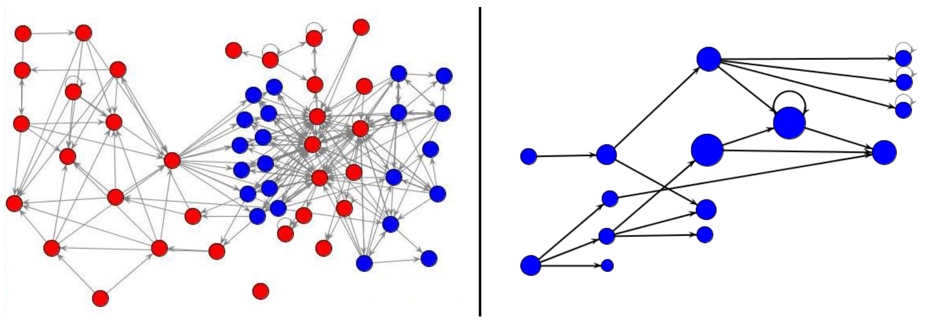


Figure 7.7: *Visualizations of a small working rule base. The left picture shows the unaltered static structure, the right picture shows the structure after rule nodes have been scaled based on their use, low level axioms have been hidden and some rules representing high-level entities have been joined.*

7.4 Implementation

The prototype of this visualization system is implemented as a standalone Java application. For the actual graph the JUNG [88] (Java Universal Network/Graph) Framework is used. The layout of the graphs is done with the Fruchterman-Reingold algorithm [65] in an implementation that is part of the JUNG framework[6] . Moving of nodes is also supported by the JUNG framework. The visualization system utilizes log files containing the prooftrees for queries and a file representing a serialization of the rule graph.

7.5 Prior Work

To the author's best knowledge there is no approach that uses usage data in the form of prooftrees for the visualization of rule bases. In fact the author is not aware of any approach that uses the runtime rule interactions to create visualizations of entire rule bases.

There is a large number of approaches that visualize the inference process that lead to a single result (e.g. [55, 24]) and some that show the static structure of rule bases [99, 72] but none that uses runtime rule interactions to create visualizations of entire rule bases. The approaches that show the static structure of rule bases also do not consider the challenges posed by large rule bases, high level editors and the hiding of rules.

Further visualizations of rules exist within the data mining community [23, 58, 77, 27, 96]. These approaches visualize association rules in order to facilitate the analysis of these rules and the extraction of the most important information from them. They also face the problem of a large set of rules from which a few interesting ones need to be selected. These approaches, however, are mostly concerned with the problems posed by the statistical nature of association rules. The goal of these visualization as well as the data used to create them, are very different from the visualizations described in this chapter.

[6]Please note that the right side of Figure 7.7 is not directly generated through this layout algorithm - for this picture some nodes were moved in order to better show the overall structure of the rule base.

7.6 Conclusion

This chapter presented a novel visualization of a rule base's structure based on the runtime interaction of rules. It can help developers understand the overall structure of the rule base that - without such a tool - is opaque to them. In this way the visualization can aid developers by making it transparent which parts of the rule base are connected; which parts of a rule base would be affected by a change.

The proposed visualization is independent of the procedural nature of the inference engine. Further, algorithms to join and hide specific rules were presented; these enable the visualization to hide low level rules, join rules that have been created through the compilation of high-level rules and to also show the interaction of modules.

The implementation of the presented visualization so far is only prototypical - integrating it into an development environment and evaluating its utility for the developer is the most pressing future work. Another possible venue for future work is the examination of this visualization for profiling - giving the developers some understanding for why a particular query takes long - and test coverage. Work in the latter direction has already started under the author's supervision and is described in [94].

Chapter 8

Conclusion

8.1 Achievements

The starting point for this thesis was the apparent contradiction between the perception of rule bases as simple to create and the experience of rule bases as hard to debug and difficult to create without faults. This contradiction was analyzed using data from a survey of developers, experiments and experiences from three rule base development projects. With this analysis ten problem areas were identified as challenges for removing or preventing faults in rule bases; challenging areas that also point to possible areas of improvement. Based on the aggregation of these areas using the developed fault lifecycle model, testing, debugging, static analysis and visualization were chosen as concrete and particularly promising approaches for tackling the debugging challenge. Each of these areas was further examined within the context of this thesis.

In the area of **testing**, a formal account of the notions of the test entities was developed, a novel test coverage measure based on least general generalization was conceived and a testing framework based on these notions was implemented and evaluated. The evaluation over more than 100 hours showed that the developed testing concepts and their implementation were usable for domain experts, very important for the domain experts' motivation, indispensable for the developers to learn rule base development and needed to create working rule bases.

To better support the **debugging** of rule bases, Explorative Debugging was proposed as a novel and purely declarative debugging paradigm for these systems. The notions of mutation extended dependency graph and mutation extended prooftree were developed as new declarative properties of rule bases that can be used for Explorative Debugging. In this context

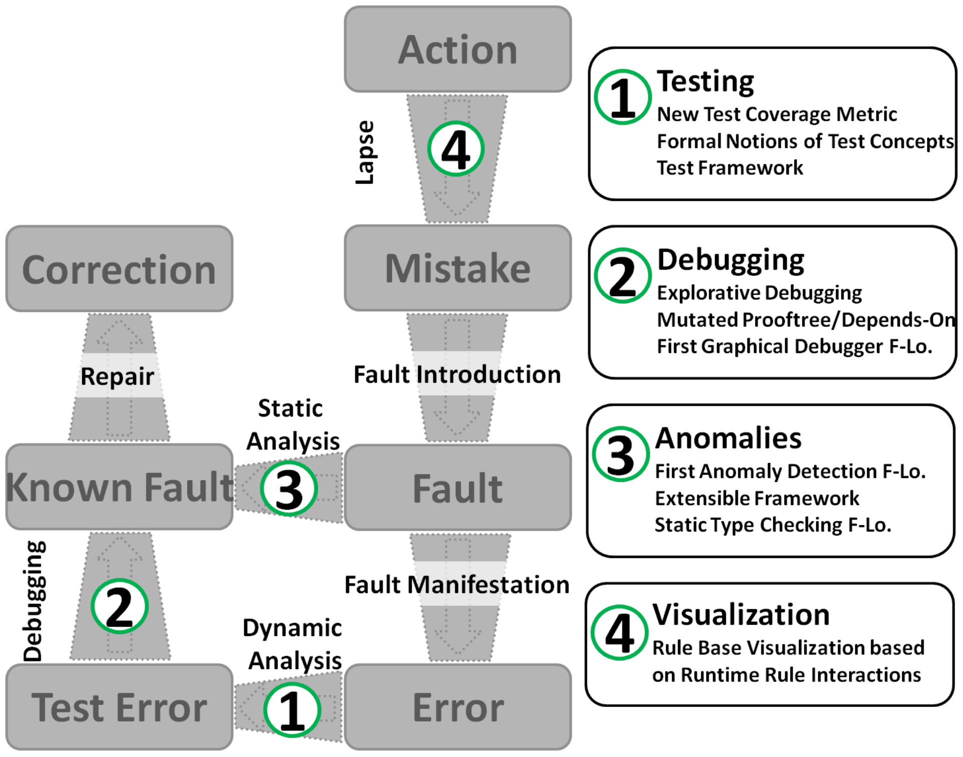


Figure 8.1: *Overview of the main contributions of this thesis*

the first ever graphical debugger for F-logic was created. Another implemented debugger is (at the time of this writing) the only open source implementation of partial and mutated prooftrees. An experiment comparing Explorative Debugging to the state of the art procedural debugging paradigm showed a significant improvement in the time needed to identify faults, while the accuracy increased.

To improve the **static analysis** for fault detection, an anomaly detection framework for F-logic was developed. This framework is the first anomaly detection system for F-logic and it includes the first implementation of static type checking for F-logic. It is mostly implemented in F-logic itself, integrated into a rule engineering environment and easily extensible.

Finally, to support users in understanding the rule base and the consequences of changes to it, a novel approach to the **visualization** of the overall structure of the rule base was developed. The novelty of this approach lies in the use of runtime rule interactions to show the overall structure of the rule base. This approach is independent of the procedural nature of the inference engine and allows to view the rule base at different levels of abstraction.

8.2 Future Work

The work in this thesis has opened up many possibilities for future work; many of which were already detailed in the conclusion section of the earlier chapters. Overall, the most promising and broad ones are: the further development of the proposed systems towards their evaluation in practical use, the extension of the approaches to support collaborative rule base engineering on the web and a further examination of the application of distance metrics for near conclusion in rule based reasoning.

This thesis has identified and experimentally evaluated many practical approaches to tackle the debugging challenge of rule based systems. However, only in large scale practical use can these concepts and tools prove whether they can in fact significantly improve fault prevention and removal during real-life rule base development; in particular the anomaly detection and visualization approaches can only be evaluated in this way. Hence, the further refinement of the created tools and the integration with rapidly evolving rule base engineering tools are important next steps.

With the partial and mutated prooftrees, the debugging chapter in this thesis discussed declarative specification for *near conclusions*; i.e. conclusion that could almost, with only slight syntactic variations, be reached. One important venue for further work is the examination of the runtime speed of these approaches and their scalability for larger rule bases; one possible improvement could be implementations that perform best-first search instead of the depth-first search of the current system. In addition the notion of near conclusion introduced with the mutated prooftree is potentially useful for areas outside of debugging; i.e. to support fault tolerant reasoning in domains with noisy rule bases where some false conclusions are acceptable.

The work presented in this thesis centers around supporting the removal and prevention of faults in rule bases created with traditional desktop applications and involving only a very small number of developers. However, the advent of large scale collaborative creation of structured data (as exemplified by Wikipedia) points the way towards a future where large, diverse and distributed groups of people jointly create knowledge bases. The focus of this work on end user developers and agile processes makes many results applicable for such a scenario; however, the large scale, permanent change and permanent inconsistency to be expected in such a system still pose many challenges.

Appendix A

Developer Survey

The section contains the questions from the developer survey in their original layout.

A.1 Rule Base Basics

Thanks for taking this survey! Please answer all questions with respect to the largest rule base in whose development you've been involved with in the past 5 years.

Sorry, the winner of the Canon SD1100 (Ixus 80is) has already been selected; though you can still participate in the survey if you like.

Which rule language and environment did you use primarily?

◇ Clips	◇ JBoss Rules / Drools	◇ SWRL - Bossam
◇ Computer Associates AION	◇ Jena Rules	◇ SWRL - KAON2
◇ Corticon Business Rules	◇ Jess	◇ SWRL - Pellet
◇ EXSYS Corvid	◇ Mandarax	◇ SWRL - RacerPro
◇ Fair Isaac Blaze Advisor	◇ Pegasystems Business Rules	◇ Versata 6 BRMS
◇ F-logic - Flora2	◇ Prolog - GNU Prolog	◇ XSB
◇ F-logic - ontoprise	◇ Prolog - SWI Prolog	◇ Yasu Quick Rules (SAP)
◇ gensym g2 (Versata)	◇ Prolog - tuProlog	
◇ Ilog JRules/.Net rules	◇ Prolog - Visual Prolog	

Other (please specify)

How large was the rule base?

Approximate person month development time for entire software:
Approximate person month development time for the rule base:
Approximate number of rules:
Approximate size of average rule (number of conditions or body-atoms):
Apptoximate size of largest rule (number of conditions or body-atoms):

How many people participated in the development of the rule base?

Rule development experts and knowledge engineers:
Other software developers:
Domain experts that created rules themselves:
Domain experts as consultants:
Domain experts for verification and validation:
Others:

How is the rule base used?

◇ Deployed (Commercial)
◇ Deployed (Research)
◇ In development, deployment planned
◇ Prototype, also used by others than the developer(s)
◇ Prototype/Experiment, only used by the developer(s)
Other (please specify):

What is the task of the rule base? (e.g. diagnosis, fraud detection, workflow management ...)

(Freeform answer)

What is the domain of the rule base (e.g. medicine, eCommerce, logistics ...)

(Freeform answer)

A.2 Development Process and Tools

Which development process was used?

◇ Did not use a specific development process
◇ Waterfall model / Specification driven
◇ Agile (e.g. ABRD, Xtreme Programming)
◇ Other Iterative and Incremental processes (e.g. RUP)
◇ Knowledge engineering (e.g. CommonKADS)
◇ Prototype driven (e.g. MOKA)
Other (please specify)

What kind of tool(s) were used to create and edit the rule base?

◇ Simple text editor
◇ Textual Rule Editor (with syntax highlighting for the rule language)
◇ Business Rule Templates / constraint natural language rule editor
◇ Graphical rule editor
◇ Spreadsheet based rule editor
◇ Decision trees rule editor
◇ Tools to learn rules from data or text
◇ An IDE that allows to edit, load, debug and run rules
Other (please specify)

What kind of verification and validation activities were performed?

◇ Testing with actual data
◇ Testing with contrieved data
◇ Testing - structural testing guided by test coverage metrics
◇ Testing - Regression testing
◇ Review by domain experts
◇ Review by developers
◇ Parallel use of system by rule expert/knowledge engineer
◇ Parallel use of system by domain expert
◇ Rule base visualization to aid review
◇ Formal verification
◇ Anomaly Detection (e.g. automatic detection of rule cycles or use of non existing classes)
Other (please specify)

What kind of tools were used for debugging?

- ◇ None
- ◇ Command line procedural debugger / tracer (specify breakpoints, investigate program state and step through program execution)
- ◇ Graphical procedural debugger / tracer (specify breakpoints, investigate program state and step through program execution)
- ◇ Algortihmic/deductive debugger (system tries to identify error by asking user for results of subcomputations)
- ◇ Explanations (system generates descriptions that explain results)
- ◇ 'Why-Not' Explanations (special explanations that can also explain why some conclusion was not reached)
- ◇ Tools that automatically change the rule base to remove errors (e.g. automatic theory revision)

Other (please specify)

What kind of bugs were encountered? What were the indications that some rule is faulty?

	never	seldom	frequent	not applicable
A query/test would not terminate	◇	◇	◇	◇
A query/test did not return any result	◇	◇	◇	◇
A wrong result was returned	◇	◇	◇	◇
A part of the result missing	◇	◇	◇	◇
The rule engine crashed	◇	◇	◇	◇
Other (please specify)				

A.3 Issues and End

What were the most important issues in the development of the rule base?

	Not an issue	Annoyance	Hindered development
Rule expressivity - could not (easily) represent what was needed	◇	◇	◇
Runtime performance - rule base too slow	◇	◇	◇
Editing of rules was hard	◇	◇	◇
Debugging was difficult	◇	◇	◇
Understanding the rule base was diffcult	◇	◇	◇
Determining the completeness of the rule base was difficult	◇	◇	◇
Developers had little experience with rule languages	◇	◇	◇
Organizing collaboration between the developers was difficult	◇	◇	◇
Maintenance - keeping the rule base up to data	◇	◇	◇
Supporting tools (rule engine, editors, ...) missing, unsuited or immature	◇	◇	◇
Other important issues			

How does the rule base and its development process compare to a ′conventional program (created with procedural/object oriented languages) of similar size?

	Rule base superior	Comparable	Conventional program superior	Don′t know
Ease of change and maintenance	⋄	⋄	⋄	⋄
Ease of creation	⋄	⋄	⋄	⋄
Ease of debugging	⋄	⋄	⋄	⋄
Reliability	⋄	⋄	⋄	⋄
Runtime performance	⋄	⋄	⋄	⋄
Tool support for development	⋄	⋄	⋄	⋄
Understandability	⋄	⋄	⋄	⋄

Your Experience

How many years of experience do you have with rule based systems:	
How many years of experience do you have with creating computer programs in general:	

Any comments, remarks about this survey?
(Freeform answer)

Enter your email address, if you like to enter the drawing of the camera. The winner will be drawn from the email addresses of all people that compeletely filled out the survey. Your address will not be shared or used to SPAM you.
(Freeform answer)

Would you like to have the results of the survey send to your email address.
⋄ yes

Appendix B

Evaluation Rule Base One

This section contains the first of the two rule bases used in the evaluation.

B.1 Rules

```
@prefix zach: <http://www.fzi.de/ipe/zach#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

# Return all grandfathers of Mike
[Query:
  (x,x,x)
<-
  (zach:Mike,zach:has_grandfather,?Grandfather)
]


# Someone is the (paternal) grandfather of someone, when someone has a
# father who has him as a father.
[Paternal_Grandfather:
  (?child, zach:has_grandfather, ?grandfather)
<-
  (?child, zach:has_father, ?father)
  (?father, zach:has_father, ?grandfather)
]


# Someone is the (maternal) grandfather of someone, when someone has a
# mother who has him as a father.
[Maternal_Grandfather:
  (?child, zach:has_grandfather, ?grandfather)
<-
  (?child, zach:has_mother, ?father)
  (?father, zach:has_father, ?grandfather)
]

# Someone is the (paternal) grandmother of someone, when someone has a
# father who has him as a mother.
[Paternal_Grandmother:
  (?child, zach:has_grandma, ?grandma)
<-
  (?child, zach:has_father, ?mother)
  (?mother, zach:has_mother, ?grandma)
]


# Someone is the (maternal) grandmother of someone, when someone has a
# mother who has him as a mother.
[Maternal_Grandmother:
  (?child, zach:has_grandma, ?grandma)
<-
  (?child, zach:has_mother, ?mother)
  (?mother, zach:has_mother, ?grandma)
]


# A ?child has a ?father, if it can be inferred that '?father has_child
# ?child'
[Has_Father:
  (?child, zach:has_father, ?father)
<-
  (?father, rdf:type, zach:Father)
  (?father, zach:has_child, ?child)
]

# A ?child has a ?mother, , if it can be inferred that '?mother has_child
# ?child'
[Has_Mother:
  (?child, zach:has_mother, ?mother)
<-
  (?mother, rdf:type, zach:Mother)
  (?mother, zach:has_child, ?child)
]
```

B.2 RDF Statements

```
@prefix : <http://www.fzi.de/ipe/zach#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
```

```
:Abraham rdf:type :Male;
  rdf:type :Father;
  :has_child:Michelle.

:Michelle rdf:type :Female;
  rdf:type :Mother;
  :has_child :Mike.

:Francis rdf:type :Male;
  rdf:type :Father;
  :has_child :Mike.

:Mike rdf:type :Male.
```

Appendix C

Evaluation Rule Base Two

This section contains the second of the two rule bases used in the evaluation.

C.1 Rules

```
# Example Rule File
@prefix zach: <http://www.fzi.de/ipe/zach#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

# Return all arguments ?B about a some apartment preferences ?B
[Query:
  (x,x,x)
<-
  (?A,rdf:type,zach:Apartment)
  (zach:Family,zach:is_argument_for,?A)
]


# If an apartement is well suited for something
# and this is a preference, then this becomes an
# argument for this apartment
[Rule_Well_Suited:
  (?B, zach:is_argument_for,?A)
<-
  (?AP,rdf:type,zach:ApartmentPreferences)
  (?AP,zach:preference,?B)
  (?A,rdf:type,zach:Apartment)
  (?A,zach:suitedFor,?B)
]
```

```
# If an apartment is well suited for kids and suited for cars,
# then it is 'suitedForFamilies'
[Rule_Suited_For_Families:
  (?Apartment, zach:suitedFor, zach:Family)
<-
  (?Apartment, rdf:type, zach:Apartment)
  (?Apartment, zach:suitedFor,zach:Kids)
  (?Apartment, zach:has_property,zach:SuitedForCars)
]

#if an apartment has a garden, then it is well suited for kids.
[Rule_Garden_For_Kids:
  (?Apartment, zach:suitedFor, zach:Kids)
<-
  (?Apartment, rdf:type, zach:Apartment)
  (?Apartment, zach:has_property,zach:Garden)
]

# If an apartment has a pool, then it is well suited for kids.
[Rule_Pool_For_Kids:
  (?X, zach:suitedFor, zach:Kids)
<-
  (?X,rdf:type,zach:Apartment)
  (?X,zach:has_property,zach:Pool)
]

# If an apartment is near a school, then it is well suited for kids.
[Rule_School_For_Kids:
  (?X, zach:suitedFor, zach:Kids)
<-
  (?X,rdf:type,zach:Apartment)
  (?X,zach:has_property,zach:NearSchool)
]
```

C.2 RDF Statements

```
@prefix : <http://www.fzi.de/ipe/zach#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

:MikesPreferences rdf:type :ApartmentPreferences;
  :preference :Family.

:BayViewApartment rdf:type :Apartment;
  :has_property :Pool;
  :has_property :SuitedForCars.
```

Appendix D

Reification Ontology

The rule reification ontology.

```
//Concepts ---------------------------------------------

   ElementOf :: Atom.
   IsA :: Atom.
   DataAtom :: Atom.
   PAtom :: Atom.
   DataType :: Atom.


   Function :: Term.
   List :: Function.

   Constant :: Term.
   NUMBER :: Constant.
   STRING :: Constant.

   Variable :: Term.

   Literal[].
   Rule[].


 //Relations ---------------------------------------------

   Number[value=>NUMBER; type=>STRING].

   String[value=>STRING].

   Constant[value=>STRING].
```

```
Variable[identifier=>STRING].

IsA[subConcept=>Term; superConcept=>Term].

Literal[atom=>Atom; isPositive=>boolean; module=>Term;
   variables=>>Term].

ElementOf[element=>Term; concept=>Term].

Atom[predicateSymbol=>STRING; arity=>integer; arguments=>List].

Function[arguments=>List; functionSymbol=>STRING;
   arity=>integer; variables=>>Term].

PAtom[arguments=>List].

DataAtom[relationName=>Term; value=>Term; object=>Term].

DataType[relationName=>Term; relationType=>Term; concept=>Term].

DataSetType[relationName=>Term; relationType=>Term; concept=>Term].

Rule[headLength=>NUMBER; bodyLength=>NUMBER; bodies=>List;
   heads=>List;identifier=>Term; module=>TERM; variables=>>Term;
   headVariables=>>Term; bodyVariables=>>Term].
```

Appendix E

Anomaly Heuristics

This appendix details rules used for the identification of anomalies. In the first part helper rules are introduced, then the actual anomaly detection rules are presented with their explanation data.

E.1 Helper Rules

```
/* ***********************************************
* root(Class,Module)
*
* Class is root class in module.
*********************************************** */
RULE root:
FORALL Concept_, TempNumber_,Module
  root(Concept_,Module)
<-
  (number_superclasses(Concept_,TempNumber_,Module)) AND
  (isnumber(TempNumber_)) AND
  (equal(TempNumber_,0)).

/* *******************************************
* number_subclasses(Concept,NumberSubclasses,Module)
*
* Well, the number of subclasses of a class
******************************************** */
RULE number_subclasses:
FORALL Concept_,NumberSubclasses_,Module
  number_subclasses(Concept_,NumberSubclasses_,Module)
<-
  class(Concept_,Module) AND
  (EXISTS Subconcept_
      directsub_(Subconcept_,Concept_)@Module AND
      count(Concept_,Subconcept_,NumberSubclasses_)).

RULE number_subclasses0:
FORALL Concept_,Module
  number_subclasses(Concept_,0,Module)
<-
  class(Concept_,Module) AND
  (NOT EXISTS Subconcept_
      directsub_(Subconcept_,Concept_)@Module).

/* *******************************************
* number_superclasses(Concept,NumberSuperclasses,Module)
*
* the number of superclasses of a class.
******************************************** */
RULE number_superclasses:
FORALL Concept_,Number_,Module
  number_superclasses(Concept_,Number_,Module)
<-
  class(Concept_,Module) AND
  (EXISTS Superconcept_
      directsub_(Concept_,Superconcept_)@Module AND
      count(Concept_,Superconcept_,Number_)).

RULE number_superclasses0:
FORALL Concept_,Module
  number_superclasses(Concept_,0,Module)
<-
  class(Concept_,Module) AND
  (NOT EXISTS Superconcept_
      directsub_(Concept_,Superconcept_)@Module).

/* ***********************************************
* var_type(Variable,Type,Rule)
*
* In the context of Rule, Variable is known to be of Type.
************************************************ */
RULE var_type:
FORALL V,T,R,A,TEMP1
   var_type(V,T,R)
<-
   type_atom(A,R) AND
   A[arguments->[variable(V),constant(T),TEMP1]]@rules_.

/* ***********************************************
* type_atom(Atom,Rule)
*
* All atoms that set the type of an variable
```

```
************************************************ */
RULE type_atom:
FORALL A,R,M,P
   type_atom(A,R)
<-
   has_atom(R,A,M) AND
   A[predicateSymbol->P]@rules_ AND
   (
      equal(P,"directisa_") OR
      equal(P,"isa_")
   ).

/* ************************************************
* att_rule(Rule, Entity, Property)
*
* Properties/Attributes used in a rule on a entity  (this may
* constant(...) or variable(...))
************************************************ */
RULE att_rule:
FORALL R,E,P,A,I,M
  att_rule(R,I,P)
<-
  att_atom(A,R) AND
  A[arguments->[I,constant(P),E,M]]@rules_.

/* ************************************************
* att_atom(Atom,Rule)
*
* All atoms that set the value of an objects property / attribute
************************************************* */
RULE att_atom:
FORALL R,A,M,P
   att_atom(A,R)
<-
   has_atom(R,A,M) AND
   A[predicateSymbol->P]@rules_ AND
   (
      equal(P,"att_") OR
      equal(P,"setatt_")
   ).

/* ************************************************
* constant_in_rule_body(RULE,CONSTANT,MODULE)
*
* All constants in the rule body
************************************************ */
FORALL R,CONSTANT,MODULE,ATOM, ARGUMENTS
    constant_in_rule_body(R, CONSTANT, MODULE)
<-
```

```
   has_body_atom(R,ATOM,MODULE) AND
   ATOM[arguments->ARGUMENTS]@rules_ AND
   inlist(constant(CONSTANT),ARGUMENTS)@rules_.

FORALL R,CONSTANT,MODULE,ATOM,FUNCTION,ARG,ARGUMENTS,FARGUMENTS
    constant_in_rule_body(R, CONSTANT, MODULE)
<-
   has_body_atom(R,ATOM,MODULE) AND
   ATOM[arguments->ARGUMENTS]@rules_ AND
   inlist(FUNCTION,ARGUMENTS)@rules_ AND
   FUNCTION:Function@rules_ AND
   FUNCTION[arguments->FARGUMENTS] AND
   equal(ARG,FARGUMENTS)@rules_ AND
   function2entity(ARG,CONSTANT).

/* ************************************************
* has_atom(Rule,Atom,Module)
*
* All atoms of rule.
************************************************ */
RULE ruleAtom:
FORALL R,A,M
  has_atom(R,A,M)
<-
  (has_body_atom(R,A,M) OR has_head_atom(R,A,M)).

/* ************************************************
* has_head_atom(Rule,Atom,Module)
*
* All body atoms for a rule.
************************************************ */
RULE ruleAtomHead:
FORALL R,A,M,BODY_EL,HEADS
  has_head_atom(R,A,M)
<-
   is_rule(R,M) AND
   R[heads->HEADS]@rules_ AND
   inlist(BODY_EL,HEADS)@rules_ AND
   BODY_EL[atom->A]@rules_.

/* ************************************************
* has_body_atom(Rule,Atom,Module)
*
* All body atoms for a rule.
************************************************ */
RULE ruleAtomBody:
FORALL R,A,M,BODY_EL,BODIES
  has_body_atom(R,A,M)
<-
```

```
   is_rule(R,M) AND
   R[bodies->BODIES]@rules_ AND
   inlist(BODY_EL,BODIES)@rules_ AND
   BODY_EL[atom->A]@rules_.

/* *********************************************
* is_rule(Rule,Module)
*
* all rules in one module
********************************************** */
RULE isRule:
FORALL R,M
  is_rule(R,M) <- R:Rule@rules_ AND R[module->M]@rules_.


/* **********************************************
* function2Entity(functionArray,Entity)
*
* The entity for the a ns_ function object
*********************************************** */
FORALL A1,A2,ENTITY
  function2entity([A1,constant(A2)], ENTITY)
<-
  equal(ENTITY,A1#A2).


/***********************************************
* entity
*
* Everything that is a class, object, property or module
********************************************* */
FORALL Thing,Module
  entity(Thing,Module)
<-
  (class(Thing,Module)) OR
  (object(Thing,Module)) OR
  (module(Thing)) OR
  (property(Thing,Module)).

/* **********************************************
* module(X)
* An approximation of the modules that exist (only those that
* contain at least one rule)
*********************************************** */
RULE modules:
FORALL R,M
  module(M) <- is_rule(R,M).

/* *********************************************
```

```
* property(Property,Module)
*
* All things defined as properties.
********************************************** */
RULE property:
   FORALL P,C,R,M
   property(P,M) <- (C[P=>R]@M or C[P=>>R]@M).


/* ******************************************
* object(Object,Module)
*
* All defined objects (things that are instances
* of something or things that have attribute
* values defined for them.
******************************************* */
RULE object1:
FORALL Instance_,Class_,Module_
  object(Instance_,Module_)
<-
  directisa_(Instance_,Class_,Module_).

RULE object2:
FORALL Instance_, Temp_, Temp2_,Module_
  object(Instance_,Module_)
<-
  Instance_[Temp_->Temp2_]@Module_.


/* ******************************************
* class(Class,Module)
*
* All defined concepts
******************************************* */
RULE conceptSubclass:
FORALL Concept_, Temp_,Module_
  class(Concept_,Module_) AND class(Temp_,Module_)
<-
  Concept_::Temp_@Module_.

RULE conceptConcept:
FORALL Concept,Module class(Concept,Module) <- Concept[]@Module.

RULE conceptAttribute:
FORALL Concept_,Temp_,Temp2_,Module_
  class(Concept_,Module_)
<-
  Concept_[Temp_=>Temp2_]@Module_.
```

```
RULE conceptISA:
FORALL Instance_, Name_, Module_
  class(Name_,Module_)
<-
  directisa_(Instance_,Name_)@Module_.

/* ********************************************
* has_defined_property
*
* is property defined in ontology
******************************************** */
RULE has_defined_property:
FORALL I,P,C,R,M
  has_defined_property(I,P,M)
<-
  I:C@M and (C[P=>R]@M or C[P=>>R]@M).

RULE has_defined_property_defaults:
FORALL C,P,R,M
  has_defined_property(C,default(P),M)
<-
  (C[P=>R]@M or C[P=>>R]@M).

/* ********************************************
* primitive_type(X)
*
* all primitive types known to flogic
********************************************* */
primitive_type(string).
primitive_type("http://www.w3.org/2001/XMLSchema"#string).
primitive_type(STRING).
primitive_type(number).
primitive_type(NUMBER).
primitive_type(integer).
primitive_type(INTEGER).
primitive_type("http://www.w3.org/2001/XMLSchema"#decimal).
primitive_type("http://www.w3.org/2001/XMLSchema"#number).
primitive_type("http://www.w3.org/2001/XMLSchema"#integer).
primitive_type("http://www.w3.org/2001/XMLSchema"#double).

/* ********************************************
* is_type_conform(X,Y)
*
* is value of instance element of type
* the type_conform_ISA rule really isn't 100% correct, but should work.
******************************************** */

// for literals
RULE type_conform_string:
  FORALL X is_type_conform(X,string) <- isstring(X) .

RULE type_conform_XMLString:
FORALL X
  is_type_conform(X,"http://www.w3.org/2001/XMLSchema"#string)
<-
  isstring(X).

RULE type_conform_STRING:
  FORALL X is_type_conform(X,STRING)
<-
  isstring(X) or isconstant(X).

RULE type_conform_number:
  FORALL X is_type_conform(X,number)
<-
  isnumber(X).

RULE type_conform_NUMBER:
  FORALL X is_type_conform(X,NUMBER)
<-
  isnumber(X).

RULE type_conform_number:
  FORALL X is_type_conform(X,integer)
<-
  isnumber(X).

RULE type_conform_NUMBER:
  FORALL X is_type_conform(X,INTEGER)
<-
  isnumber(X).

RULE type_conform_XMLDecimal:
  FORALL X
  is_type_conform(X,"http://www.w3.org/2001/XMLSchema"#decimal)
<-
  isnumber(X).

// for literals, although in fact ontobroker does not support
// integer semantics, now.
RULE type_conform_XMLDecimal:
FORALL X
  is_type_conform(X,"http://www.w3.org/2001/XMLSchema"#integer)
<-
  isnumber(X).

RULE type_conform_XMLDecimal:
FORALL X
```

```
  is_type_conform(X,"http://www.w3.org/2001/XMLSchema"#double)
<-
  isnumber(X).

//Actually does not exist in XMLSchema, but is used
RULE type_conform_XMLNumber:
FORALL X
  is_type_conform(X,"http://www.w3.org/2001/XMLSchema"#number)
<-
  isnumber(X).

RULE type_conform_ISA:
  FORALL V,R,M is_type_conform(V,R)
<-
  V:R@M.

/* ************************************************
* Serverity level
************************************************ */
severity_error.
severity_warning.
severity_information.
```

E.2 Anomaly Heuristics

```
<anomalyHeuristics>
<fLogicHeuristics>

<flogicAnomalyHeuristic>
  <title>Property Undefined for Variable in Rule</title>

  <!-- A rule uses a property on a instance and this instance
  if not known to be of a type that defines this property-->

  <id>property_undefined_for_variable</id>
  <category>Rule</category>
  <performanceIndex>0</performanceIndex>
  <defaultOn>true</defaultOn>
  <code>
    RULE property_undefined_for_variable:
    FORALL R,E,P,M
      problem(property_undefined_for_variable,severity_warning,M,[R,P,E])
    <-
      is_rule(R,M) AND
      att_rule(R,variable(E),P) AND
      property(P,M) AND
      NOT (
        EXISTS T,RANGE
        var_type(E,T,R) AND
        (T[P=>RANGE]@M or T[P=>>RANGE]@M)
      ).
  </code>

  <description> An attribute or relation is used in a rule on an
    instance that does not have the attribute or relation. Run
    this test after you removed attributes or relations from the
    ontology that some rules may still use.
  </description>

  <explanation> The rule $0$ uses the attribute/relation $1$ on
    variable $2$, but $2$ is not known have this attribute/relation.
    This is only a warning - the rule may still work.
  </explanation>

</flogicAnomalyHeuristic>


<flogicAnomalyHeuristic>

  <title>Rule Property Unknown</title>
  <id>rule_property_unknown</id>
  <category>Rule</category>
  <performanceIndex>1</performanceIndex>
  <defaultOn>true</defaultOn>

  <code>
    RULE rule_property_unknown:
      FORALL R,P,M,E
      problem(rule_property_unknown,severity_error,M,[R,P])
    <-
      is_rule(R,M) AND
      att_rule(R,E,P) AND
      NOT property(P,M).
  </code>

  <description> A rules uses an attribute or relation that
    is not defined anywhere. Run this test after you
    removed attributes or relations from the ontology that
    some rules may still use.
  </description>

  <explanation>The rule $0$ uses the property $1$ - such
    a property is not defined for any concept! This rule
    will not work.
  </explanation>
```

```
</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Range Concept Undefined</title>
  <id>range_concept_undefined</id>
  <category>Properties</category>
  <performanceIndex>1</performanceIndex>
  <defaultOn>true</defaultOn>

  <code>
    RULE range_concept_undefined1:
    FORALL P,C,R,M
      problem(range_concept_undefined,severity_error, M,[C,P,R])
    <-
      directatttype_(C,P,R)@M AND NOT(class(R,M)
      OR primitive_type(R)).

    RULE range_concept_undefined2:
    FORALL P,C,R,M
      problem(range_concept_undefined,severity_error, M,[P,C,R])
    <-
      directsetatttype_(C,P,R)@M AND NOT(class(R,M)
      OR primitive_type(R)).
</code>

  <description> An error is created for each relation that goes to
    a concept that no longer exist. Run this test after you
    removed concepts from the ontology.
  </description>

  <explanation> The property $0$ of concept $1$ goes to the concept
    $2$ the (no longer) exists. Maybe you replaced the concept with a
    different one and you need to replace it in the definition of the
    relation as well? Maybe the relation is no longer needed and can
    be deleted?
  </explanation>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Rule Body Constant Undefined</title>
  <id>body_constant_undefined</id>
  <defaultOn>true</defaultOn>
  <category>Rule</category>
  <performanceIndex>1</performanceIndex>
  <code>
    RULE body_constant_undefined:
    FORALL R,C,M
      problem(body_constant_undefined,severity_warning,M,[R,C])
    <-
      constant_in_rule_body(R,C,M) AND
      NOT entity(C,M).
</code>

  <description> A rule uses an instance in the rule body that is
    not defined elsewhere. Run this test after you removed instances
    from the ontology that may still be used in rules.
  </description>

  <explanation> The rule $0$ uses the instance $1$ that is not
    defined anywhere. Maybe you replaced the instance in the ontology
    and need to replace it in the rule as well? With the instance no
    longer there, can the rule be removed as well?
  </explanation>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Head Variable Unbound</title>
  <id>head_var_unbound</id>
  <category>Rule</category>
  <performanceIndex>0</performanceIndex>
  <defaultOn>false</defaultOn>

  <code>
    RULE head_var_unbound:
    FORALL R,HEAD_VAR,Module
      problem(head_var_unbound,severity_error, Module,[R,HEAD_VAR])
    <-
      R:Rule@rules_ AND R[module->Module]@rules_ AND
      R[headVariables->variable(HEAD_VAR)]@rules_
      AND NOT R[bodyVariables->variable(HEAD_VAR)]@rules_.
  </code>

  <description> An error is created for each rule that uses a variable
    in the head but not in the body. This is a fatal modelling error that
    will stop a rule from working. This will not happen, if you use only
    the rule editor to create rules.
  </description>

  <explanation>The rule $0$ uses the variable $1$ in the head but not in
    the body. This error means that this rule can’t work and that any
    attempt to use it in a inference will case an exception. Correct this
    exception by either removing the variable from the head or adding
    it to the body.
  </explanation>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Many subclasses</title>
  <id>style_manySubclasses</id>
  <category>Style</category>
  <performanceIndex>1</performanceIndex>
  <defaultOn>true</defaultOn>

  <description>A warning is generated for each concept with more
    than $num$ subclasses. A large number of subclasses is often
    considered a bad practice. Run this test to refine a large
    ontology for example before declaring it finished.
  </description>

  <explanation>$0$ has $1$ subclasses. More than $num$ is usually
    considered a bad practice. Maybe some of the subconcepts can
    be grouped under some new concept?
  </explanation>

  <code>
  FORALL Concept_,NumberSubclasses_, Module
    problem(style_manySubclasses,severity_warning,Module,
      [Concept_,NumberSubclasses_])
  <-
    number_subclasses(Concept_,NumberSubclasses_,Module) AND
    greater(NumberSubclasses_,$num$).
  </code>

  <parameters>
    <parameter>
      <defaultValue>12</defaultValue>
      <id>num</id>
      <name>Number Subclasses</name>
      <description>The number of subclasses that are allowed
        before a warning gets created.
      </description>
    </parameter>
  </parameters>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>
```

```
  <title>Few subclasses</title>
  <id>style_fewSubclasses</id>
  <category>Style</category>
  <performanceIndex>1</performanceIndex>
  <defaultOn>true</defaultOn>

  <description> A warning is generated for each concept with less
    than $num$ subconcepts. A very small number of subclasses is
    often considered a bad practice. Run this test to refine a
    large ontology for example before declaring it finished.
  </description>

  <explanation>$0$ has $1$ subclasses. Less than $num$ is usually
    considered a bad practice. Maybe you meant to add more
    subconcepts later? Would it be possible to remove some subconcept
    of $1$ and add its subconcepts directly?
  </explanation>

  <code>
    FORALL Concept_,NumberSubclasses_,Module
      problem(style_fewSubclasses,severity_warning,Module,
        [Concept_,NumberSubclasses_])
    <-
      number_subclasses(Concept_,NumberSubclasses_,Module) AND
      greater(NumberSubclasses_,0) AND
      less(NumberSubclasses_,$num$).
  </code>

  <parameters>
    <parameter>
      <defaultValue>2</defaultValue>
      <id>num</id>
      <name>Number Subclasses</name>
      <description>The minimum number of subclasses. </description>
    </parameter>
  </parameters>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Class Leaf</title>
  <id>class_leaf</id>
  <defaultOn>true</defaultOn>
  <category>Concepts</category>
  <performanceIndex>1</performanceIndex>

  <code>
    FORALL Concept_, Module
```

```
      problem(class_leaf,severity_warning,Module,[Concept_])
    <-
      class(Concept_,Module) AND
      (NOT EXISTS TempConcept_ TempConcept_::Concept_@Module) AND
      (NOT EXISTS TempInstance_ TempInstance_:Concept_@Module).
  </code>

  <description> A warning is created for each concept that has
    neither instances nor subconcepts. This often indicates that
    the author forget to create instances. Run this test regularly
    to check the ontology for loose ends.
  </description>

  <explanation>$0$ is a concept that has neither instances nor
    subconcepts. This may be intentional (for example when
    representing a taxonomy) but propably indicates missing instances.
  </explanation>

</flogicAnomalyHeuristic>


<flogicAnomalyHeuristic>

  <title>Class Undefined</title>
  <id>class_undef</id>
  <defaultOn>true</defaultOn>
  <category>Concepts</category>
  <performanceIndex>1</performanceIndex>

  <code>
    FORALL Instance_, Concept_, Module
      problem(class_undef,severity_warning, Module,[Instance_,Concept_])
    <-
      Instance_:Concept_@Module AND
      (NOT EXISTS Superconcept_ Concept_::Superconcept_@Module) AND
      (NOT EXISTS Subconcept_ Subconcept_::Concept_@Module) AND
      (NOT EXISTS Attribute_,Type_ Concept_[Attribute_=>Type_]@Module)
      AND (NOT Concept_[]@Module).
</code>

  <description> A warning is created for each concept that is used in
    a isa statement but not defined elsewhere. Run this test after
    you removed concepts from the ontology.
  </description>

  <explanation>$0$ is an instance of $1$, but $1$ is not defined
    elsewhere. This is not necessary a problem, but propably indicates
    that you meant a different concept or deleted the concept in the
    meantime.
  </explanation>

</flogicAnomalyHeuristic>


<flogicAnomalyHeuristic>

  <title>Cardinality Constraint Violation</title>
  <id>cardinality_constraint_hurt</id>
  <category>Instances</category>
  <performanceIndex>3</performanceIndex>
  <defaultOn>false</defaultOn>

  <code>
    RULE cardinality_constraint_hurt:
    FORALL Instance,Concept,Property,Range,Value1,Value2,Module
      problem(cardinality_constraint_hurt,severity_error,Module,
        [Instance,Property,Concept])
    <-
      Instance:Concept@Module AND Concept[Property=>Range]@Module AND
      Instance[Property->Value1]@Module AND
      Instance[Property->Value2]@Module AND not equal(Value1,Value2).
  </code>

  <description> An error is created for instances that have more than
    one property value for properties defined with cardinality 0..1.
    Run this test to refine a large ontology for example before
    declaring it finished. Start it before going to lunch, because it
    may take a long time.
  </description>

  <explanation>$0$ has more than one different values for relation or
    attribute $1$ but the schema defines a maximum cardinality of one
    for its concept $2$.
  </explanation>

</flogicAnomalyHeuristic>

<flogicAnomalyHeuristic>

  <title>Instance Property Undefined</title>
  <id>instance_property_undefined</id>
  <category>Instances</category>
  <performanceIndex>3</performanceIndex>
  <defaultOn>false</defaultOn>

  <code>
    RULE instance_property_undefined:
    FORALL Module,Instance, Property, Value
    problem(instance_property_undefined,severity_error,
      Module,[Instance,Property,Value])
  <-
    Instance[Property->Value]@Module AND NOT
    has_defined_property(Instance,Property,Module).
</code>

  <description> An error is created for attributes or relations of
    instances that is not defined. Run this test after removing
    attributes or relations. Start it before going to lunch, because
    it may take a long time.
  </description>

  <explanation>$0$ has the property $1$ (with value $2$). This property
    is not defined for the concepts of $0$.
  </explanation>

</flogicAnomalyHeuristic>


<flogicAnomalyHeuristic>

  <title>Instance Range Type</title>
  <id>instance_range_type</id>
  <defaultOn>false</defaultOn>
  <category>Instances</category>
  <performanceIndex>3</performanceIndex>

  <code>
    RULE instance_range_type:
    FORALL Module, Instance, Method, Value , Range, Concept
    problem(instance_range_type,severity_error, Module,
      [Instance, Method, Value,Range])
    <-
      (Concept[Method=>Range]@Module OR
      Concept[Method=>>Range]@Module) AND
      Instance:Concept[Method->Value]@Module AND
      NOT is_type_conform(Value,Range).
</code>

  <description> An error is created for attributes or relations of
    instances that have the wrong type. Run this test after you
    changed the type range type of properties.
  </description>

  <explanation>$0$ has the attribute or relation $1$ with value $2$.
    This value is not compatible to the specified range "$3$" for
    this property.
  </explanation>
```

```
</flogicAnomalyHeuristic>


<flogicAnomalyHeuristic>
  <title>Root Concepts</title>
  <id>info_rootconcept</id>
  <defaultOn>true</defaultOn>
  <category>Concepts</category>
  <performanceIndex>1</performanceIndex>

  <code>
    FORALL Concept_,Module
      problem(info_rootconcept,severity_information, Module,[Concept_])
    <-
      root(Concept_,Module).
</code>

  <description> An information is generated for each concept that
    has no superconcepts. Run this test to refine a large ontology
    for example before declaring it finished.
  </description>

  <explanation>$0$ is a root concept. This is not a problem, but may
    indicate that you forgot to specify a superclass.
  </explanation>

</flogicAnomalyHeuristic>
</fLogicHeuristics>

</anomalyHeuristics>
```

Bibliography

[1] H. Alvestrand. Rfc 3066 - tags for the identification of languages. Technical report, IETF, 2001.

[2] J. Angele. How to write F-logic programs. Technical report, Ontoprise GmbH, 2005. http://www.ontoprise.de/fileadmin/user_upload/Publications_EN/Tutorial_FLogic_en.pdf.

[3] J. Angele and G. Lausen. Ontologies in F-logic. In S. Staab and R. Studer, editors, *Handbook on Ontologies*. Springer, 2004.

[4] J. Baroff, R. Simon, F. Gilman, and B. Shneiderman. Direct manipulation user interfaces for expert systems. In *Expert systems: the user interface*, pages 99–125. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[5] V. Barr. Rule-based system testing with control and data flow techniques. In *Proceedings of the International Software Quality-Week*, 1996.

[6] V. Barr. Applications of rule-base coverage measures to expert system evaluation. In *Proceedings of the AAAI*, pages 411–416, 1997.

[7] W. Beaton and J. d. Rivieres. Eclipse platform technical overview. Technical report, The Eclipse Foundation, 2006. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[8] M.Y. Becker and S. Nanz. The role of abduction in declarative authorization policies. In *in Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2008.

[9] D. Beckett. New syntaxes for RDF. Technical report, Institute for Learning and Research Technology, Bristol, 2004.

[10] D. Beckett. RDF/XML syntax specification (revised). Technical report, W3C Recommendation, 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[11] D. Beckett. Turtle - terse RDF triple language. http://www.dajobe.org/2004/01/turtle/, 2004. (accessed 2007-03-30).

[12] B. Belli and O. Jack. Implementation-based analysis and testing of prolog programs. In *ISSTA '93, Proceedings of the 1993 ACM SIGSOFT international symposium on software testing and analysis*, 1993.

[13] F. Belli and Oliver Jack. A test coverage notion for logic programming. In *Proceddings of the Sixth International Symposium on Software Reliability Engineering*, pages 133–142, 1995.

[14] R. Fielding T. Berners-Lee and L. Masinter. Rfc 2396 - uniform resource identifiers (URI): Generic syntax. Technical report, IETF, 1998.

[15] T. Berners-Lee. Notation 3. Technical report, W3C, 1998. http://www.w3.org/DesignIssues/Notation3.

[16] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[17] A. Malhotra P.V. Biron. XML schema part 2: Datatypes second edition. Technical report, W3C Recommendation, 2004. http://www.w3.org/TR/xmlschema-2/.

[18] B. Boehm. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR, 2000.

[19] H. Boley, M. Kifer, P.-L. Pa, and A. Polleres. Rule interchange on the web. In *Reasoning Web*, pages 269–309. Springer, 2007.

[20] P.A. Bonatti, D. Olmedilla, and J. Peer. Advanced policy explanations on the web. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, 2006.

[21] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental evaluation of verification and validation tools on martian rover software. In *Proceedings of the SEI Software Model Checking Workshop*, 2003.

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

[23] D. Bruzzese and P. Buono. Combining visual techniques for association rules exploration. In *Proceedings of the working conference on advanced visual interfaces*, pages 381 – 384, Gallipoli, Italy, 2004.

[24] A. Bösel, T. Linke, and T. Schaub. Profiling answer set programming: The visualization component of the nomore system. In *Logics in Artificial Intelligence*, pages 702–705. Springer, 2004.

[25] B.G. Buchanan and E.H. Shortliffe, editors. *Rule-Based Expert Systems*. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Publishing Company, 1984.

[26] A. Bundy, H. Pain, P. Brna, and L. Lynch. A proposed Prolog story. research paper 283. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1985.

[27] M. Burch, S. Diehl, and P. Gerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 37 – 46, 2005.

[28] L. Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the Workshop on Logic Programming*, 1980.

[29] J. J. Carroll and G. Klyne. Resource description framework (RDF): Concepts and abstract syntax. Technical report, W3C, 2004. http://www.w3.org/TR/rdf-concepts/.

[30] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. Technical report, Hewlett Packard, 2003.

[31] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

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

[33] C.L. Chang, J.B. Combs, and R.A. Stachowitz. A report on the expert systems validation associate (EVA). *Expert Systems with Applications*, 1:217–230, 1990.

[34] V.K. Chaudhri, B.E. John, S. Mishra, J. Pacheco, B. Porter, and A. Spaulding. Enabling experts to build knowledge bases from science textbooks. In *K-CAP '07: Proceedings of the 4th international conference on Knowledge capture*, pages 159–166, New York, NY, USA, 2007. ACM.

[35] S. Chaw, P. Porter, K. Barger, and P. Yeh. Towards an ontology-independent problem solver. Technical report, AI Lab, University of Texas at Austin, 2007. http://www.cs.utexas.edu/ftp/pub/AI-Lab/tech-reports/UT-AI-TR-07-340.pdf.

[36] D. Chester. The translation of formal proofs into english. *Artificial Intelligence*, 7:261–278, 1976.

[37] W.J. Clancey. The epistemology of a rule based expert system - a framework for explanation. *Artificial Intelligence*, 20, 1983.

[38] W.J. Clancey. Use of MYCIN's rules for tutoring. In *Rule-based expert systems*. Addison-Wesley, 1984.

[39] P. Clark, S.-Y. Chaw, K. Barker, V. Chaudhri, P. Harrison, J. Fan, B. John, B. Porter, A. Spaulding, J. Thompson, and P. Yeh. Capturing and answering questions posed to a knowledge-based system. In *K-CAP '07: Proceedings of the 4th international conference on Knowledge capture*, pages 63–70, New York, NY, USA, 2007. ACM.

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

[41] S. Craw and D. Sleeman. Knowledge based refinement of knowledge based systems. Technical report, The Robert Gordon University, 1996. http://www.scms.rgu.ac.uk/publications/95/95_2.ps.gz.

[42] M. Cusumano, A. MacCormack, C., F. Kemerer, and B. Crandall. Software development worldwide: The state of the practice. *IEEE Software*, 20, 2003.

[43] J. d. Bruijn, E. Franconi, and S. Tessaris. Logical reconstruction of rdf and ontology languages. In *Proceedings of the PPSWR 2005*, pages 65–71, 2005.

[44] J.-M. Marc David, J.-P. Krivine, and B. Ricard. Building and maintaining a large knowledge-based system from a "knowledge level" perspective: the diva experiment. In *Second generation expert systems*, pages 376–401. Springer-Verlag, 1993.

[45] R. Davis. *Teiresias: Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of large Knowledge Bases*. PhD thesis, Computer Science Department, Stanford University, CA, 1976.

[46] R. Davis. Interactive transfer of expertise. *Artificial Intelligence*, 12:121–157, 1979.

[47] R. Davis. Interactive transfer of expertise. In *Rule-based expert systems*. Addison-Wesley, 1984.

[48] J. de Bruijn, E. Franconi, and S. Tessaris. Logical reconstruction of normative RDF. 2005.

[49] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In *DS-8*, pages 351–369, 1999.

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

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

[52] N. Dershowitz and Y.-J. Lee. Deductive debugging. In *SLP*, pages 298–306, 1987.

[53] M. Ducassé and J. Noy. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19(20):351–384, 1994.

[54] Eclipse. Zest: The Eclipse visualization toolkit. http://www.eclipse.org/gef/zest/, 2008. (accessed 2008-05-10).

[55] M. Eisenstadt and M. Brayshaw. The transparent prolog machine (tpm): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.

[56] M. Eisenstadt, M. Brayshaw, and J. Paine. *The Transparent Prolog Machine: visualizing logic programs*. Intellect Books, 1991.

[57] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy theory revision: Induction and abduction in INTHELEX. *Machine Learning*, 38(1-2):133–156, 2000.

[58] U. Fayyad, G. Grinstein, and A. Wierse. *Information Visualization in Data Mining and Knowledge Discovery*. Morgan Kaufmann, 2001.

[59] D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker: The very high idea. In *Proceedings of the 11th International Flairs Conference (FLAIRS-98), Sanibal Island, Florida*, 1998.

[60] Eclipse Foundation. Rich client platform. http://wiki.eclipse.org/index.php/Rich_Client_Platform, 2007. (accessed 2007-03-21).

[61] N. Friedland, P. Allen, M. Witbrock, G. Matthews, N. Salay, P. Miraglia, J. Angele, S. Staab, D. Israel, V. Chaudhri, B. Porter, K. Barker, and P. Clark. Towards a quantitative, platform-independent analysis of knowledge systems. In *Proceedings of the Conference on Knowledge Representation and Reasoning - KR-2004*, pages 507–515. AAAI Press, JUN 2004.

[62] N.S. Friedland, P.G. Allen, G.Matthews, M.J. Witbrock, D.Baxter, J.Curtis, B.Shepard, P.Miraglia, J. Angele, S. Staab, E. Mönch, H. Op-

permann, D. Wenke, D.J. Israel, V.K. Chaudhri, B.W. Porter, K. Barker, J. Fan, S. Yi Chaw, P.Z. Yeh, D. Tecuci, and P. Clark. Project halo: Towards a digital aristotle. *AI Magazine*, 25(4):29–48, 2004.

[63] J. Frohn, R. Himmeroder, P. Kandzia, and C. Schlepphorst. How to write F-logic programs in FLORID: A tutorial for the database language F-logic. Technical report, Institut Für Informatik, Universitat Freiburg, Germany, 1996.

[64] J. Frohn, R. Himmeroder, P.-Th. Kandzia, G. Lausen, and C. Schlepphorst. Florid: a prototype for f-logic. page 583, 1997.

[65] T.M.J. Fruchterman and E.M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[66] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceeding of the Fifth Logic Programming Symposium*, pages 1070–1080, 1988.

[67] A. Ginsberg. *Automatic Refinement of Expert System Knowledge Bases*. Morgan Kaufmann Publishers, 1988.

[68] A. Ginsberg. Knowledge base reduction: A new approach to checking knowledge bases for inconsistency and redundancy. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 88)*, pages 585–589, 1988.

[69] Ontoprise GmbH. Project halo interview transcripts - internal report. Technical report, Ontoprise GmbH, 2006.

[70] C. Golbreich and A. Imai. Combining swrl rules and owl ontologies with protege owl plugin, jess and racer. In *Proceedings of the 7th International Protege Conference*, 2004.

[71] J. Manuél Gómez-Pérez, M. Erdmann, and M. Greaves. Applying problem solving methods for process knowledge acquisition, representation, and reasoning. In *K-CAP*, pages 15–22, 2007.

[72] C.A.McK. Grant. *Software Visualization in Prolog*. PhD thesis, University of Zurich, 2001.

[73] J. Grant and D. Becket. RDF test cases - n-triples. Technical report, W3C Recommendation, 2004. http://www.w3.org/TR/rdf-testcases/#ntriples.

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

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

[76] D. Hamilton and K. Kelley. State-of-the-practice in knowledge-based system verification and validation. *Expert Systems With Applications*, 3:403–410, 1991.

[77] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 2000.

[78] S. Handschuh and A. Maedche. Cream - creating relational metadata with a component-based, ontology-driven annotation framework. In *Proceedings of the First International Conference on Knwoledge Capture (K-CAP)*, 2001.

[79] D. Hansch and M. Erdmann. Deep authoring, answering and representation of knowledge by subject matter experts - quantitative analysis of intermediate evaluation. Technical report, ontoprise GmbH, 2006.

[80] D. Hansch and M. Erdmann. Deep authoring, answering and representation of knowledge by subject matter experts - study to improve reasoning performance for question answering in darkmatterstudio. Technical report, ontoprise GmbH, 2006.

[81] D. Hansch and M.Erdmann. Deep authoring, answering and representation of knowledge by subject matter experts - final report implementation phase. Technical report, ontoprise GmbH, 2006.

[82] W. Harrison. The dangers of end-user programming. *IEEE Software*, July/August:5–7, 2004.

[83] B. McBride P. Hayes. RDF semantics. Technical report, WC3 Recommendation, 2004. http://www.w3.org/TR/rdf-mt/.

[84] IEEE Std 610.121990, IEEE standard glossary of software engineering terminology. Technical report, IEEE Computer Society, 1990.

[85] Ilog. Ilog business rule management systems. http://www.ilog.com/products/businessrules/, 2008. (accessed 2008-29-04).

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

[87] W.L. Johnson. Agents that learn to explain themselves. In *AAAI*, pages 1257–1263, 1994.

[88] JUNG. Jung - java universal network/graph framework. http://jung.sourceforge.net/, 2008. (accessed 2008-08-03).

[89] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1992.

[90] R.A. Kowalski A.C. Kakas and F. Tony. The role of abduction in logic programming. In C.J. Hogger D.M. gabbay and J.a. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1998.

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

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

[93] J.D. Kiper. Structural testing of rule-based expert systems. *Transactions on Software Engineering and Methodology*, 1:168–187, 1992.

[94] F. Kleiner. Diploma thesis: Testvollständigkeit für F-logic Wissensbasen. Technical report, FZI Forschungszentrum Informatik an der Universität Karlsruhe (TH), 2006.

[95] G.E. Krasner and S.T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[96] M. Kreuseler and H. Schumann. A flexible approach for visual data mining. In *IEEE Transactions on Visualization and Computer Graphics*, volume 8, pages 39 – 51, 2002.

[97] C. Larman and V.R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, June:47–56, 2003.

[98] J.C. Lester and B.W. Porter. Developing and empirically evaluating robust explanation generators: The knight experiments. *Computational Linguistics*, 23(1):65–101, 1997.

[99] J.W. Lewis. An effective graphics user interface for rules and inference mechanisms. In *Conference on Human Factors in Computing Systems Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 139 – 143, Boston, Massachusetts, United States, 1983.

[100] J.W. Lloyd. *Foundations of Logic Programming (2nd Extended Edition)*. Springer-Verlag, 1987.

[101] J.W. Lloyd and R.W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[102] A. MacCormack. Product-development practices that work. *MIT Sloan Management Review*, pages 75–84, 2001.

[103] A. Maedche and V. Zacharias. Clustering ontology-based metadata on the semantic web. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 348 – 360, 2002.

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

[105] M.T. Maybury. Communicative acts for explanation generation. *International Journal of Man-Machine Studies*, 37(2):135–172, 1992.

[106] M.T. Maybury. Planning multimedia explanations using communicative acts. In M.T. Maybury and W. Wahlster, editors, *Readings in Intelligent User Interfaces*. Morgan Kaufmann Publishers Inc, 1998.

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

[108] D.L. McGuinness and P.P. da Silva. Explaining answers from the semantic web: The inference web approach. *Journal of Web Semantics*, 1(4), 2004.

[109] M. Mehrotra. Rule groupings: a software engineering approach towards verification of expert systems. Technical report, NASA Contract NAS1-18585, Final Rep., 1991.

[110] D. Merrit. Best practices for rule-based application development. *The Architecture Journal*, 2004.

[111] B.S. Miller and M. Sprenger. *Explanation abilities in Computer as Assistants: A New Generation of Support Systems*, pages 68–80. Lawrence Erlbaum Associates, 1996.

[112] R.J. Money and B.L. Richards. First-order theory revision. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 447–451, 1991.

[113] K. Morik, B.-E. Kietz, W. Emde, and S. Wrobel. *Knowledge Acquisition and Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[114] C. Mues and J. Vanthienen. *Efficient Rule Base Verification Using Binary Decision Diagrams*, pages 445–454. 2004.

[115] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[116] B. Myers, A. Ko, and M. Burnett. Invited research overview: End-user programming. In *ACM Conference on Human-Computer Interaction (CHI'06)*, 2006.

[117] G.J. Myers. *Software Reliability - Principles and Practices*. John Wiley & Sons, New York, 1976.

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

[119] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D.Pecora. Checking an expert systems knowledge base for consistency and completeness. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI 85)*, pages 375–378, 1985.

[120] N. Noy and D. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical report, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, 2001.

[121] E. Nyberg and T. Mitamura. The KANTOO machine translation environment. In *AMTA*, pages 192–195, 2000.

[122] ObjectConnections. Common knowledge. `http://www.objectconnections.com/default.html`, 2007. (accessed 2007-02-13).

[123] D.E. O'Leary. *Design, Development and Validation of Expert Systems: A Survey of Developers*, pages 3–18. John Wiley & Sons Ltd., 1991.

[124] Ontoprise. Ontostudio. `http://www.ontoprise.de/`, 2007. (accessed 2007-02-27).

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

[126] H. Pain and A. Bundy. What stories should we tell novice prolog programmers. In R. Hawley, editor, *Artificial Intelligence programming environments*. Ellis Horwood, 1987.

[127] A. Paschke, J. Dietrich, A. Giurca, G. Wagner, and S. Lukichev. On self-validating rule bases. In *Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, 2006.

[128] C. Peirce. *Philosophical Writings of Peirce, Chapter Abduction and Induction*, pages 150–156. Dover Publications, 1955.

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

[130] W. Perry. *Effective Methods for Software Testing*. John Wiley and Sons, New York, 1996.

[131] T. Pfirrmann. Diplomarbeit - logikbasierte Modellierung von Anforderungen and Software Lösungen. Technical report, Institut für Angewandte Informatik und Formale Beschreibungsverfahren der Universität Karlsruhe, 2006.

[132] G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 1969.

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

[134] A.D. Preece, R.D. Bell, and C.Y. Suen. Verifying knowledge-based systems unsing the COVER tool. In *Proceedings of the 12th IFIP Congress*, pages 231–237, 1992.

[135] A.D. Preece, S. Talbot, and L. Vignollet. Evaluation of verification tools for knowledge-based systems. *International Journal Human-Computer Studies*, 47(5):629–658, 1997.

[136] L. De Raedt. *Interactive theory revision: an inductive logic programming approach*. Academic Press Ltd., London, UK, UK, 1992.

[137] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 119–140. Plenum Publications, 1978.

[138] WOLFRAM Research. Mathematica. `http://www.wolfram.com/products/mathematica/index.html`, 2008. (accessed 2008-08-01).

[139] M. B. Rosson, J. Balling, and H. Nash. Everyday programming: Challenges and opportunities for informal web development. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 2004.

[140] S. Ruggieri. Decidability of logic program semantics and applications to testing. *The Journal of Logic Programming*, 46:103–137, 2000.

[141] RuleML. The rule markup initiative. `http://www.ruleml.org/`, 2008. (accessed 2008-08-03).

[142] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *IEEE Software*, 23:82–90, 2006.

[143] J. Rushby and J. Crow. Evaluation of an expert system for fault detection, isolation, and recovery in the manned maneuvering unit. NASA contractor report CR-187466. Technical report, SRI International, 1990.

[144] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 0002 edition, 2003.

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

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

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

[148] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *L/HCC'05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[149] A. Carlisle Scott, William J. Clancey, Randell Davis, and Edward Shortliffe. Methods for generating explanations. In *Rule-based expert systems*. Addison-Wesley, 1984.

[150] K. Seer. The 2005 business rules awareness survey. *Business Rule Journal*, 2005.

[151] E.Y. Shapiro. *Algorithmic program debugging*. PhD thesis, Yale University, 1982.

[152] E.H. Shortliffe. *MYCIN: A rule-based computer program for advising physcians regarding antimicrobial therapy selection*. PhD thesis, Stanford University, 1974.

[153] E.H. Shortliffe, R. Davis, S.G. Axline, B.G. Buchanan, C.C. Green, and S.N. Cohen. Computer-based consultations in clinical therapeutics: Explanation and rule acquisition capabilities of the mycin system. *Computers and Biomedical Research*, 8:303–320, 1975.

[154] E.H. Spafford and C. Viravan. Experimental designs: Testing a debugging oracle assistant. Technical report, SERC-TR-120-P, Software Engineering Research Center, Purdue University, West Lafayette, 1992.

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

[156] Y. Sure, J. Angele, and S. Staab. Ontoedit multifaceted inferencing for ontology engineering. *Journal on Data Semantics*, LNCS 2800:128–152, 2003.

[157] SurveyMonkey. Surveymonkey. `http://www.surveymonkey.com/`, 2008. (accessed 2008-05-29).

[158] M. Suwa, A.C. Scott, and E.H. Shortlifee. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 1982.

[159] W. R. Swartout. Xplain: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21(3):285–325, 1983.

[160] SWI-Prolog. Xpce/swi-prolog — graphical tracer front-end. `http://www.swi-prolog.org/guitracer.html`, 2007. (accessed 2007-02-13).

[161] SWT. The standard widget toolkit. `http://www.eclipse.org/swt/`, 2007. (accessed 2007-02-18).

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

[163] Visual Prolog. `http://www.visual-prolog.com/`. (accessed 2008-08-19).

[164] A. van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 18:620–650, 1991.

[165] F. van Harmelen. Applying rule-base anomalies to KADS inference structures. *Working Notes from IJCAI-95 Workshop on Verification and Validation of Knowledge-Based Systems*, 21:271—280, 1995.

[166] W. van Melle, E.H. Shortliffe, and B.G. Buchanan. Emycin: A knowledge engineer's tool for constructing rule-based expert systems. In *Rule-based expert systems*. Addison-Wesley, 1984.

[167] D. A. Waterman. *Machine Learning of Heuristics*. PhD thesis, Stanford University, 1968.

[168] R. Westman and P. Fritzson. Graphical user interfaces for algorithmic debugging. In *In Automated and Algorithmic Debugging*, pages 273–286. SpringerVerlag, 1993.

[169] J. Wielemaker. SWI-Prolog reference manual. Technical report, 1997. http://gollem.science.uva.nl/cgi-bin/nph-download/SWI-Prolog/refman/refman.pdf.

[170] J. Wielemaker. An overview of the swi-prolog programming environment. Technical report, University of Amsterdam, The Netherlands, 2003.

[171] D.C. Wilkins. *Knowledge base refinement as improving an incorrect and incomplete domain theory*, pages 493–513. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

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

[173] C. Yang and M. Kifer. Flora-2 user manual. Technical report, Department of Computer Science, Stony Brook University, 2002.

[174] G. Yang, M. Kifer, and C. Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *CoopIS/DOA/ODBASE*, pages 671–688, 2003.

[175] V. L. Yu, D.G. Buchanan, E.H. Shortliffe, E.H. Wraith, S.D. Davis, R. Scott, and A.C. Cohen. Evaluating the performance of a computer-based consultant. *Computer Programs in Biomedicine*, 95(1), 1979.

[176] V. Zacharias. Visualization of rule bases - the overall structure. In *Submitted to the 7th International Conference on Knowledge Management - Special Track on Knowledge Visualization and Knowledge Discovery*, 2007.

[177] V. Zacharias. The debugging of rule bases. In *to appear in Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*. IGI Global, 2009.

[178] L. Zhao and S. Elbaum. A survey on quality related activities in open source. *SIGSOFT Software Engineering Notes*, 25(3):54–57, 2000.

[179] L. Zhao and S. Elbaum. Quality assurance under the open source development model. *Journal of Systems and Software*, 66:65–75, 2003.