Explorative Debugging for Rapid Rule Base Development

Valentin Zacharias and Andreas Abecker

FZI Research Center for Information Technologies, Haid-und-Neu Strasse 10-14,
Karlsruhe 76131, Germany
zacharias|abecker@fzi.de

Abstract. We present Explorative Debugging as a novel debugging paradigm for rule based languages. Explorative Debugging allows truly declarative debugging of rules and is well suited to support rapid, try-and-error development of rules. We also present the Inference Explorer, an open source explorative debugger for horn rules on top of RDF.

1 Debugging Semantic Web Rules

Semantic Web rule languages can be an important tool for the rapid development of Semantic Web applications [7]. The large scale use of these languages is, however, currently still hampered by missing tool support for their creation - in particular missing debugging support.
Debugging support for Semantic Web rule bases must address two challenges:

- It is known that web developers have a particularly high percentage of end user programmers [9, 6]; hence development tools for the Semantic Web have to take special care to adjust to end user programmers. For debugging tools this means in particular to support the try and error, rapid, incremental development process often observed with end user programmers [10, 11].
- Rules are declarative programs that describe what is true but not how something is calculated. A debugger must take this into account.

This paper starts with a definition of rules and an introduction into the rule syntax used throughout this paper. It then discusses existing debugging approaches and their limitations. After these have been established, it proposes Explorative Debugging as a better debugging paradigm. It describes the building blocks of Explorative Debugging and shows how they are implemented in the Inference Explorer application.

2 Terminology

To date there is no agreed upon definition or specification of rules for the Semantic Web, also the ideas presented in this paper can be applied to a number of different rule languages. However, in the interest of having a consistent terminology throughout this paper, we define a simple rule language based on the

rules used by the well known Jena framework [3] that also forms the basis of the implementation. In this paper only rules that work on data in the form of RDF [1] triples are considered.

A rule base is a finite set of rules. The rules are of the form:

```
Rule_Example:
   (?A,b,c) <- (?A,x,y) (?A,v,w)
```

The part to the left of the ← symbol is the *goal* or *head* of the rule, the part to the right the *body* or *subgoals*. The head of the rule consists of one *atom*, the body of a conjunction of atoms. An atom is a *triple pattern*, a triple of *terms*[1]. A triple pattern is a RDF triple without anonymous nodes but with named variables instead of some terms. A triple pattern is *matched* against an RDF triple by replacing the variables with terms from the triple; the matching succeeds when the variables can be replaced in a way that makes triple pattern and triple identical. A rule is said to *fire* when all atoms in the body of a rule match a triple, with all occurrences of one named variable replaced by the same term.

Rule bases are declarative specifications - they describe *what* is true / can be calculated, not *how* it should be done. A rule base can be evaluated in a number of different ways (e.g. top-down evaluation, bottom-up evaluation, magic sets algorithm) but the result does not depend on the kind of evaluation and rules can be created and understood without knowing the actual evaluation strategy[2].

3 Current Debugging Support For Rules

Existing debugging support for rule based systems can be divided into two groups: procedural debugging and computer controlled debugging.

All deployed debugging tools for rule based programs known to the authors are based on the procedural or imperative debugging paradigm. This debugging paradigm is well known from the world of procedural and object oriented 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. Of course a rule base does not define an order of execution - hence the order of debugging is based on the evaluation strategy of the inference engine. The steps then, are things like: the inference engine tries to prove a goal A or it tries to find more results for another goal B. This kind of debuggers for rule based systems are available as purely textual

[1] Jena also allows extended triple patterns containing functors and built-ins as atoms, however they are not currently supported in the Inference Explorer; we postpone their discussion until the future work section

[2] Note that this is only partly true for Prolog that mixes procedural and declarative aspects

tracers [12], with a simple graphical user interface [12] or integrated into graphical knowledge acquisition tools [8]. The most sophisticated of these systems was probably the Transparent Prolog Machine [4].

Procedural debuggers break the declarative programming paradigm: they force the user to learn about the working of the inference engine and encourage her to think about the program in an imperative way; also procedural debugging is very difficult to use for anything but simple top-down inference engines. This kind of debugger was necessary for languages like Prolog that mix procedural and declarative aspects; the rule languages proposed for the Semantic web, however, are purely declarative and hence can be debugged in a declarative way.

In computer controlled debugging[3] an algorithm directs the debugging process, the user only needs to answer questions about the expected outcome. The amount of research in this direction is too large to give an overview here; however, in general it can be said:

- These debuggers do not empower the programmer to learn more about the program. Many approaches just propose a change without giving the programmer a chance to test and correct her expectations.
- These debuggers can utilize only information encoded in the program, not the domain knowledge and intuition of the programmer.
- To our best knowledge none of these debuggers has ever proven to work good enough to be used outside of research.

The conclusion then is that current debugging support is unsuited for the rapid development of rules by end user programmers: It either works on the level of the inference engine, becoming hard to use and breaking the declarative programming paradigm. Or it takes control away from the programmer, disempowering the user and becoming imprecise.

4 Explorative Debugging

We propose Explorative Debugging as a better debugging paradigm for rule-based systems. Explorative Debugging works on the declarative semantics of the program and lets the user navigate and explore the inference process. It enables the user to use all her knowledge to quickly find the problem and to learn about the program at the same time. An Explorative Debugger is not only a tool to identify the cause of an identified problem but also a tool to try out a program and learn about its working.

Explorative Debugging puts the focus on rules. It enables the user to check which inferences a rule enables, how it interacts with other parts of the rule base and what role the different rule parts play. Unlike in procedural debuggers the

[3] The authors use computer controlled debugging as a broad term to refer to the areas of Algorithmic Debugging, Declarative Debugging, Automatic Debugging, Why-Not Explanation, Knowledge Refinement, Automatic Theory Revision and Abductive Reasoning

debugging process is not determined by the procedural nature of the inference engine but by the user who can use the logical/semantic relations between the rules to navigate.

An explorative debugger is a rule browser that:

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

5 Building Blocks Of Explorative Debugging

The main building blocks of Explorative Debugging for rules are: rules, rule firings, rule parts, depends-on-connections and prooftrees. These concepts describe declarative properties of the rule base and take the place of the breakpoints and different stepping commands of procedural debugging. The following sections describe each of these concepts and its role in more detail.

5.1 Rules

The main element for an explorative debugger for rule bases is rules. The debugger is always focused on one rule and can be opened for any rule. Navigation elements allow navigating between rules. Centering a debugger for rule bases around rules sounds self evident, but is not realized in most existing debuggers for rule based systems (these debuggers use the goal the inference engine currently tries to prove as main element).

5.2 Rule Firings

Rule firings are the inferences a rule currently enables. Consider the following rule:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)
```

and data[4]:

```
  :Abraham  rdf:type  :male;
            :parent   :John.
```

[4] We use the Turtle [2] syntax to represent RDF data, for briefness we omit the prefix declarations

```
:Johanna  rdf:type  :female;
          :parent   :John.

:George   :parent   :Michele.
```

In this example there is one rule firing: variable A bound to Abraham and B bound to John; with the current data the rule enables to infer that Abraham is of type father. More formally a rule firing is a set of variable bindings that satisfies the rule body.

The rule firings are the most basic form of checking whether a rule performs according to the expectations of the user. In this example the user probably expected the rule to also infer that George is a father, but sees that this is not the case. Other concepts described below will support the user in investigating this further. Rule firings should be available to the user all the time while creating rules to give quick feedback.

The *current data* - the triples used to calculate the variable bindings - can come from a default data store or from a test setup. For many simple cases the user will have just one example of the data she wants to process with the rules and use this for tests. In such cases this is the default data that can be used to automatically calculate the variable bindings without further ado. In more complex cases a programming environment must support a way to create test cases that contain the appropriate test data that can be used. The rule firings are then displayed with respect to this test data.

5.3 Rule Parts (Firings)

The natural way to examine the unexpected behavior of a rule is to look at the rule parts and to calculate the rule firings for a rule only consisting of a subset of the parts. When, like in the example above, the rule isn't firing for an expected value the user can investigate which rule atom/triple pattern is causing the problem. In this example the user may look at the variable bindings that satisfy the triple pattern (?A,rdf:type,:male), seeing that the expected George isn't included in the list of matching triples; in this way she has narrowed the problem down further.

Rule part firings allow the user to select only a part of the rule body and to look at the variable bindings that satisfy the selected triple patterns.

5.4 Depend-On Connections

The concepts described above only consider a rule and its result as an isolated entity, without the connections to other rules. Depends-on connections are links between rules that indicate that two rules seem to be able to work together, built on top of each other. Consider the following rule base:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)
```

```
Rule_Parent:
   (?X,:parent,?Y) <- (?Y,:chilt,?X)

Rule_Employer:
   (?X,rdf:type,employer) <- (?X, :owns, ?C) (?A,:employed_at,?C)
```

It can be seen that, with the right data, the rule Father could work on the result of the rule Parent: rule Parent could deduce that A is a parent of B from a child relation and the Father rule could then deduce that A is of type father. In such a case we say that rule Father *depends-on* rule Parent. The depends-on connections are calculated on the rules without consideration for the actual data that is available: a depends-on connection exists independently of the test data. The rule Employer in the example rule base has no depends-on connection to any of the other rules - there exists no set of RDF triples such that the rule Employer could directly work on triples inferred by one of the other two rules.

Formally a depends-on connection exists from rule A to rule B when rule A has a body atom that can be unified with the head atom of B; when rule A has a triple pattern in the body that matches the triple pattern in the head of rule B. Depends-on connections can also be calculated between rule parts and rules: a rule part is a subset of body atoms of a rule. A rule part then depends-on a rule and when one of its atoms is unifiable with the head of the rule.

The depends-on connections calculated this way are heuristic approximations to the rule interactions intended by the user. It is not possible to perfectly identify only those rule interactions that happen with real world data: there can be depends-on connections between rules that never interact in real life.

The importance of the depends-on connection lies in their ability to aid the user in tracking down an error across multiple rules, even if the error prevents any actual interaction between the rules from happening. The following example should illustrate this:

```
:Abraham  rdf:type  :male.
:John     :child :Abraham.
```

With the rule base above and these two RDF triples the user expects that rule Father fires for Abraham and John. Opening the debugger for rule Father, however, she sees that the rule doesn't fire. Selecting rule parts she can identify that the pattern (?A,:parent,?B) is causing the problem. This, however, is not the root problem and the depends-on connection supports the user in finding the root cause. Even though there is no actual interaction between the rules a system can show that this rule part depends on rule Parent. The user can use this connection to navigate to the other rule and to discover the root cause: the typo *chilt* in the body of rule Parent. The depends-on links may look like a minor navigation tweak in such a small rule base but becomes an important navigation instrument for large rule bases where the user may not even know about all rules that could be relevant for the current triple pattern.

Obviously there are also errors that impact the depends-on connection: had rule Parent had a typo in the head, there would be no depends-on connection. But the absence of a depends-on link is also an important clue for debugging: a user knowing that a part should depend-on a particular rule has already narrowed down the problem to a mismatch between the current rule part and the head of the other rule. A user expecting a depends-on connection to some rule may be supported by the system in finding rules that almost match.

The importance of depend-on connections cannot easily be overstated for the debugging of rule based systems. Each rule in itself is usually good to understand and a lot of research went into good visualizations for single rules. On the other hand rule interactions and the overall structure of a rule program are usually not readily visible and are often not displayed at all. The depends-on connections and the prooftrees described in the next section will often be the only way to view this structure.

5.5 Prooftrees

Prooftrees represent the actual interactions between rule and RDF facts that lead to results / rule firings. Prooftrees are a tool to quickly identify the reason for unexpected variable firings; for too many results. An example should further clarify this:

```
Rule_Father:
   (?A,rdf:type,:father) <- (?A,rdf:type,:male) (?A,:parent,?B)

Rule_Parent:
   (?X,:parent,?Y) <- (?Y,:child,?X)

Rule_ChildFromRaises:
   (?X,:child,?Y) <- (?Y,:raises,?X)

   :JohnsUncle  rdf:type  :male;
                :raises   :John.
```

Checking the rule firings for the rule "Father" the user is surprised to find that JohnsUncle is inferred to be of type father. A prooftree (or derivation tree) is a representation of the inference process that lead to a particular result and that can aid the user in such a case to find the part of the reasoning chain that acted in an unexpected way. A graphical representation of the actual prooftree for the conclusion (JohnsUncle, rdf:type, father) is shown in Figure 1. It shows that the rule Father concluded this triple and that it depended on the fact that JohnsUncle is of type male and the result of rule Parent, this in turn on another rule and on the triple (JohnsUncle, raises, John). The prooftree allows the user to quickly understand the whole inference process and to pinpoint the rule ChildFromRaises as the root cause of the problem.

The prooftree represents the logical/semantic relationships between rules/ facts and is independent of the actual evaluation algorithm of the inference

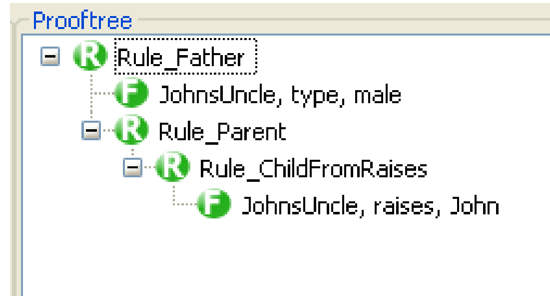

Fig. 1. *Prooftree for the triple (JohnsUncle, rdf:type, father).*

engine. When a rule instantiation in the prooftree is listed as dependent on a fact it says that this fact is necessary for this prooftree to exist. The current result will not be concluded, unless this fact is present[5] The inference engine algorithm can be said to search for and discover the prooftrees, but the prooftrees themselves are defined on the semantics of rules.

6 The Inference Explorer

The Inference Explorer is a debugger for RDF rules that implements the explorative debugging paradigm and that integrates all the building blocks for truly declarative debugging described in the section before.

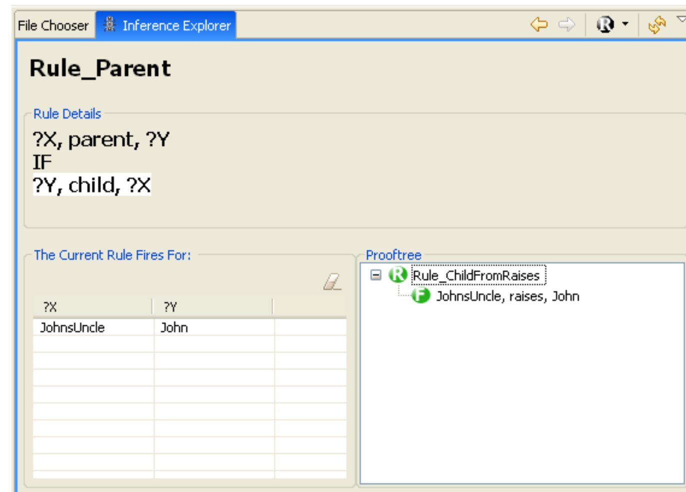

Fig. 2. *User interface of the Inference Explorer*

[5] Unless, of course, there are multiple prooftrees for one result

The user interface of the Inference Explorer (see Figure 2) consists of three main areas: the rule details in the top, the result view bottom left and the details view in the bottom right. Both the result view and the details view change depending on what the user has selected.

The rule details view shows a textual description of the currently selected rule. The user can select parts of the rules to get more information about these and the rule parts are shown in different colors to highlight unsatisfiable triple patterns.

The result view to the bottom left of the debugger shows the rule firings, the values that satisfy the rule body. When the user selects a rule part in the rule details view the result view changes to display the variable bindings that satisfy this rule part.

The details view to the bottom left changes greatly in response to the currently selected element in the debugger.

- When nothing or a rule part is selected, the details view shows the depends-on connections between the current rule/rule part and other rules in the rule base. The user can click on any rule listed there to open it in the debugger.
- When a result line is selected in the result view, the details views shows the prooftree for this result. This configuration is shown in the screenshot in Figure 2. The user can click on any rule in the prooftree to open the debugger for it.

On the top of the debugger are navigation elements to navigate to arbitrary rules based on their names and elements to move *back* and *forward*, akin to similar buttons in browsers. Other buttons allow to hide URI's before the hash symbol and to reload all data and rules from the (possible changed) files. A second view allows to select the RDF and rules files used.

The InferenceExplorer is currently implemented as a stand-alone Eclipse-RCP [5] program. For RDF storage and rule parsing it uses the Jena Framework. For inferences it uses or own Try![6] inference engine - a simple backward chaining inference engine specifically created to support the debugging of rules. It forfeits speed and memory footprint in order to be easy to understand and to have internal data structures that can be translated to give a human understandable representation of the inference engine's state. Its is created to also support more complex debugging applications beyond the Inference Explorer presented here.

The InferenceExplorer and the Try! inference engine are available as open source and can be downloaded from http://code.google.com/p/trie-rules/.

7 Programming Embedded Debugging - Future Work

Traditionally, programming and debugging are seen as clearly separate activities: some part of the program is written, a test is created, run and finally the running test may be debugged. The concepts of Explorative Debugging, however, allow

[6] The complete name is TRIE - the Transparent RDF Inference Engine

to naturally integrate some part of the debugging activity into programming and can thereby facilitate faster feedback cycles.

The prerequisite for programming embedded debugging is the definition of *default data*. Default data is some set of RDF triples that is used to evaluate the rules against. This does not exclude the possibility to have other sets of test data, but there needs to be a default set.

With default data it is possible to automatically open the debugger for a rule while it is created by the user. As soon as the user creates a syntactically correct rule the debugger can evaluate this rule in the background and display to the user what this rule infers, given the other rules and the default data. Giving the user an immediate feedback on her rule as it is created.

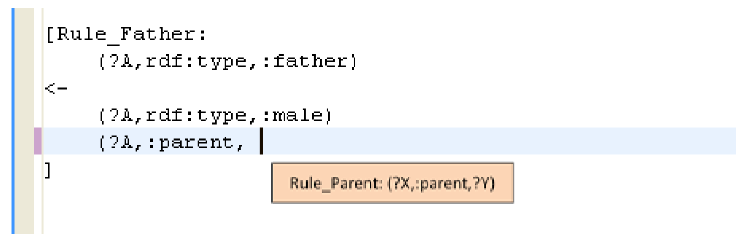

Fig. 3. *Mock-up of the integration of depends-on connections into a rule editor.*

Another concept of Explorative Debugging that can be utilized during programming are the depends-on connections. A sophisticated rule editor could immediately display the rules a triple pattern could depend-on as it is created. The mock-up in figure 3 shows how that may look like for a text based rule editor. While the user creates a body atom the one rule she is alerted to a matching head pattern in a different rule. Such integration of the depends-on connections allow the user to quickly get feedback on her assumptions about likely interactions between the rules she is creating.

8 Conclusions and Future Work

The concepts of rules, rule parts, rule firings, depends-on connections and proof-trees allow the debugging of rules purely on the declarative level. The novel debugging paradigm Explorative Debugging brings them together to allow debugging that truly takes into account the declarative nature of rules. Explorative Debugging is well suited to support short feedback loops and to support iterative development.

Our implementation of the Explorative Debugging ideas, The Inference Explorer, is the most advanced debugger for RDF rules available.

For the future we plan to extend the Try! inference engine and to integrate the debugger with editors for RDF and rules. Currently the Try! inference engine is

very limited in that it does not support functors, built-ins and anonymous node - we plan to add support for these in the coming weeks. Another problem are rule cycles that can lead the inference engine to run infinitely - we will add algorithms to detect and explain them to the user. After that the further development will be based on feedback from users of the debugger.

References

1. D. Beckett. RDF/XML syntax specification. W3C recommendation, RDF Core Working Group, World Wide Web Consortium, 2004.
2. D. Beckett. Turtle - terse RDF triple language. http://www.dajobe.org/2004/01/turtle/, 2004. (accessed 2007-03-30).
3. 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.
4. M. Eisenstadt, M. Brayshaw, and J. Paine. *The Transparent Prolog Machine: visualizing logic programs*. Intellect Books, 1991.
5. Eclipse Foundation. Rich client platoform. http://wiki.eclipse.org/index.php/Rich_Client_Platform, 2007. (accessed 2007-03-21).
6. W. Harrison. The dangers of end-user programming. *IEEE Software*, July/August:5–7, 2004.
7. M. Kifer, J. de Bruijn, H. Boley, and D. Fensel. A realistic architecture for the semantic web. In *RuleML*, pages 17–29, 2005.
8. ObjectConnections. Common knowledge. http://www.objectconnections.com/, 2007. (accessed 2007-02-13).
9. 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.
10. 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.
11. 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.
12. J. Wielemaker. An overview of the SWI-Prolog programming environment. Technical report, University of Amsterdam, The Netherlands, 2003.