Exploiting Usage Data for the Visualization of Rule Bases

Valentin Zacharias[1] and Imen Borgi[1]

FZI: Research Center for Information Technologies, Karlsruhe 76131, Germany,
{zacharias, borgi}@fzi.de

Abstract. Recent developments have made it clear that the Semantic Web will not only consist of RDF data and OWL ontologies but also of knowledge formulated as rules, hence the visualization of rules will also be an important part of user interfaces for the Semantic Web.
In this paper we describe novel ideas and their prototypical implementation for the visualization of rule bases. In the creation of the visualization our approach considers not only the structure of a rule base but also records of its usage. We also discuss the challenges for visualization algorithms posed by rule bases created with high level knowledge acquisition tools. We describe the methods we employ to deal with these challenges.

1 Introduction

1.1 The Semantic Web, Rules and their Visualization

Envisioned as a central part of the Semantic Web since its early days [5][11][22][7] rule languages lately received more attention . It has even been argued that the Semantic Web stack should be changed to give rules a more prominent role, putting rule languages on an equal footing with OWL[19]. Even though the exact definition of a Semantic Web rule language is still subject of much debate [18][4][2], it seems certain that rule languages will soon become an important pillar of the Semantic Web. Its is also safe to say that exchanging rule models on a global scale will increase the need for tools that help people understand rule bases. Sharing and adapting - main mottos of the Semantic Web - are all but unthinkable without tools that aid people in understanding the rule bases they should reuse. For these reason we see a renewed need to visualize rule bases.

F-logic, the rule based language used throughout this paper, is not one of the main contenders to become the Semantic Web rule language. It is, however, very similar to many approaches under discussion - so most of the things discussed in this paper can easily be adapted to these other rule languages. For our work F-logic had the advantage of the availability of two mature inference engines [8][29] and the ontology engineering environment Ontostudio. Furthermore, we believe that the problems related to knowledge bases created with high level visual editors discussed in this paper will appear in many cases where modern rule editors get created and used to construct knowledge bases.

1.2 Motivation for this Work

A good visualization of a rule base should quickly convey its meaning. It should enable the user to understand the scope of the rule base and point her to the central parts. It should enable the user to asses the suitability of a rule base for a particular task. It could support the user in evaluating the quality of the rule base and could aid her in correctly executing changes to it. The visualization should only

display important information, it should not confuse the user with details she doesn't need to know to achieve the goals described above.

We found simple visualization approaches to fail by these criteria. In particular visualizing rules without considering their usage led to the display of many '"phantom" dependencies between rules. While these dependencies existed in theory, we knew that a meaningful knowledge base would never include the facts that would make them relevant. Displaying these dependencies could make sense in a debugging or validation scenario - but it only confuses when the goal is to explain the rule base. This proved to be a big problem since these connections often outnumber the important dependencies by far.

Another problem occurred when high level knowledge engineering tools where used to create (parts) of the knowledge base. The content the user creates with these tools is translated into the logic language before any inference takes place. In our system this is even a two step translation: first the graphical representation is translated into F-logic, then F-logic is transformed into a logic program in normal logic. The use of such tools makes visualizing a knowledge base more difficult because there may be a considerable difference between the contents of the knowledge base and the entities the user works with and knows about.

1.3 Our Approach

The approach we are proposing utilizes usage information of the knowledge base as information source to identify the relative importance of rules and typical interaction patterns among them. We also use meta knowledge about the structure of the knowledge base to hide low level details from the user.

This paper starts with an introduction to F-logic and a discussion of what we understand as usage data of a (F-logic) knowledge base. We then describe a simple example rule base that we will use to illustrate our ideas in the rest of the paper. We continue with a discussion of the challenges posed by high level tools for the creation of knowledge bases. We then show a baseline visualization of our example knowledge base and describe the steps our system takes to transform this into a better representation. We conclude with a discussion of related work and possible directions for further research.

2 Preliminaries

2.1 F-logic

F-logic[20] or Frame Logic was developed as an object oriented approach to first order logic. It defines syntax and semantic of an object oriented deductive database. Main features of F-logic include object identity, complex objects, inheritance, polymorphic types, query methods and encapsulation. F-Logic has a sound and complete resolution based proof theory. A number of F-logic implementations exist that, while having much in common, still differ in crucial details. Hence it is important to note that for this paper we use the F-logic implementation by Ontoprise. Discussing the differences to the other versions is beyond the scope of this publication.

In this paper we will only present and explain a short F-Logic example adopted from [1], readers interested in learning more about F-logic may consult [3][20][1][12].

```
/* facts */
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[father->abraham]].
```

The first part of this example consists of a set of facts to indicate that some people belong to the classes man and woman respectively. The first section also gives information about the father and mother relations between them. According to the object-oriented paradigm, relationships between objects are modelled by method applications, e.g. applying 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. Evaluating these rules in a bottom-up way, new relationships between objects, denoted by the method "son" are added to the object base. 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 about all women and their sons, whose father is Abraham.

Before their evaluation F-logic programs are transformed into normal logic programs (horn logic with negation) with function symbols. The details of this transformation are beyond the scope of this paper (the interested reader may consult [3][20]). It is sufficient to say that this process may replace F-logic rules with more than one internal rule. Even more importantly, some core features of the language (e.g. everything related to inheritance) are realized by adding a number of rules to the knowledge base. These *axioms* are not directly visible to users of F-logic but still central to its working.

2.2 Normal Logic Programs

A *normal logic program* is a finite set of rules of rules. The rules are of the form

$$p(A) \leftarrow q(B), \mathbf{not}\, r(C, d)$$

The part to the left of the $\leftarrow$ 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 *literals*. The arguments of the predicates that make up head and body of the rule are called *terms*. Variables and constants as well as function symbols with terms as arguments are terms. A term without arguments is called a *ground term*. A rule A is said to be dependent on another rule B when an atom from the body of A *unifies* with the head atom of B. A atom unifies with another atom when the predicate symbol is identical and all their argument terms unify. A term unifies with another term when:

- the top function symbols are identical and their arguments unify
- at least one of the terms is a variable
- both are identical ground terms.

The rule graph for a rule base is a directed graph $G_R = (R, A)$ with

- A set of vertices R, there is one vertice for each rule in the rule base.
- A set of ordered pairs A of vertices called arcs or arrows. An arc $e = (x, y)$ is considered to mean that the rule represented by the vertice x depends on the rule represented by y. Such an arc exists for every rule dependency.

2.3 Usage Data and Prooftrees

Usage data is created by any query that is posed against the rule base. This query may have been created as test for the knowledge base during its creation, as competency question during the specification or sometimes during its use. The visualization is going to be better the more queries are available and the more representative for the usage of the rule base these queries are. Please note that a query will normally be posed to the rule base and a number of facts. It is common that the facts change between queries. For example a knowledge base containing diagnostic rules for diseases will also contain facts desribing the current patients. These facts will obviosly be changed between queries.

During the evaluation of a query the inference engine creates a *prooftree forest*. The prooftree forest contains one *prooftree* for each result that was returned. The prooftree stores information about the rules that where used in creating the result. 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 where directly needed to prove the query. The children of these rules are again the rules needed to prove them. The leafs of the prooftree are formed by the facts in the knowledge base. Due to the nature of the well-founded semantics[14] used to evaluate F-logic the prooftree is not always a tree but may contain circles. A bit more formally we can say that a prooftree is a directed graph $G_P = (V, A)$ with

- A set of vertices V. Each vertice can be understood as the application of one rule during the inference process. There is also (partial[1]) function $f_r : V \rightarrow R$ that gives the rule that was applied for a vertice in the prooftree.
- A set of arcs A. An arc $e = (x, y)$ is considered to mean that the application of a rule represented by vertice x depended on the rule firing represented by y.

Usage data U is the multiset of all known prooftrees for a rule base: $U = \{G_{P,1}, G_{P,2}..., G_{P,i}\}$ and the function f_r defined earlier.

3 An Example Rule Base

We created a toy example that we will use throughout the paper to illustrate our visualization ideas. This knowledge base consists of a number of rules for the diagnosis of faults in sewers. The rule base works on facts representing a number of sewers, their properties and a number of observations for these. It contains three rules representing mathematical relations between properties of sewers of different shapes. All other rules are integrating data from the observations and diagnostic data inferred by other rules for high level diagnosis and a priority classification. An example for the inferences that can be done with this rule base is the following: One rule concludes that there is a high priority problem with a sewer if it is large and has a large hole. Other rules are used to infer *large sewer* based on the throughput and *large hole* based on the observations. One rule would be able to conclude the throughput based size and shape. All in all the knowledge base for our example contains 15 rules, the ontology 40 concepts.

[1] The prooftree also contains nodes that represent the "firing" of the query and the usage of facts

The usage data for this example is made up from 15 queries. For each query there is also a situation description - a *query setup*. This setup contains the facts that describe a number of sewers, their properties and some observations. From executing these queries we get 26 prooftrees that form the input to the visualization tool. There are more prooftrees than queries because some queries return more than one result.

4 High Level Knowledge Editors

Our example was created using a new version of Ontoprise's ontology engineering program Ontostudio. One purpose of this tool is to make ontology engineering more accessible for people that are not training logicians but rather experts in the field that is to be modelled. Towards this goal it employs visual editors that shield the user from some of the complexities of creating a knowledge base. In addition F-logic is already an abstraction on top of the entities manipulated by the inference engine; it is translated before any queries are processed. These two things conspire to make visualizing the knowledge base more difficult because there is a considerable mismatch between the entities known to the user and the artifacts in the knowledge base and the prooftrees. A visualization cannot be calculated directly on the entities known to the user, because the semantics of the rule languages are not defined at this level. Which entities depend on each other, how a query is answered is only defined through the translation to a normal logic program. To do a visualization similar to the one proposed in this paper on the level of the entities known to the user would mean in essense to create another inference engine that works at this level. We will now further illustrate mismatch between user entities and knowledge base entities by looking at the handling of mathematical relationships and inheritance.

Consider the relationship between the length of the sides of a rectangle and its surface area. It is easily expressible in one equation that enables a human to calculate any of the three values provided that the other two are known. Such a relationship is, however, not directly translatable into the rigid IF-THEN structure of horn logic rules. The straightforward way to translate this into a logic programming language is to create three rules, one for each property of the rectangle. Ontostudio allows the user to specify mathematical relations and handles their translation into F-logic automatically and transparent to the user (albeit using a mechanism a bit more sophisticated than the one described above). This of course results in the knowledge base containing more rules than just the ones known by the user.

Another source of mismatch between the user view and the contents of the knowledge base is the translation of F-logic into a normal logic program. We have already stated that some parts of the F-logic that appear to the user as primitive operations are, in fact, realized by adding further rules to the knowledge base. Consider these two rules and one fact:

```
IF anything(?X) THEN ?X is a stainless_steel_sewer
IF ?A is a steel_sewer THEN something(?A)
stainless_steel_sewer subconcept steel_sewer.
```

Its intuitively obvious that the second rule could depend on the first one. But for the inference engine this connection is not visible, in particular after everything has been translated into a normal logic program.

```
isa_(?X,stainless_steel_sewer) <- anything(?X)
something(?A) <- isa_(?A,steel_sewer)
sub_(stainless_steel_sewer,steel_sewer)
```

In Ontoprise's inference engine the semantics of inheritance are realized by adding a number of *axioms* to the knowledge base. These axioms are very generic rules that make these connections. An example for an axiom would be: IF ?A is a ?Concept and ?Concept is a subconcept of ?Superconcept THEN ?A is a ?Superconcept. Again this leads to many rules being added to the knowledge base that should be kept hidden from the user but that are indispensable for its functioning.

All in all we end with a knowledge base containing 54 F-Logic rules - up from the 15 (high level) rules the user created. Such a large difference between the entities manipulated by the user and the contents of a knowledge base is still uncommon in today's knowledge engineering tools. We believe, however, that with the increasing use of graphical tools for knowledge editing and more interest in knowledge acquisition tools that can be handled by domain experts, this kind of problem will appear in more and more systems. On the one hand it is true that rules are usually a more natural expression for knowledge than for example assembler. On the other hand simplicity and runtime consideration must be taken into account when designing rule languages. The knowledge artifacts that you would like to have from a usability point of view are often not the same as the ones that can be handled easily and fast by an inference engine. In the past trained knowledge engineers bridged this gap, translating everything they heard from the domain experts into the logic language. In a future where domain experts are creating knowledge bases with ever decreasing reliance on knowledge engineers more and more of this translation needs to be done automatically.

5 Naive Rule Base Visualization

In the absense of a defined control flow or objects and their relationships, the most salient feature of rule bases are the rules and their "'depends-on"' relationships as defined in section 2.2. These relationships show how the parts of the knowledge base work together to answer queries. This relationships forms a directed graph that also forms the basis for our visualization. The nodes in the graph are the rules from the knowledge base. We have used different colors to distinguish rules created and edited by the user (blue nodes) from axioms (red nodes). Each node is labeled with the name of the rule it is representing. The arcs between the nodes represent dependencies between these rules. A rule A is said to dependend on rule B, if the head of B unifies with an atom of the body of A. In other words, the body of rule A depends on the result of rule B. Rule names are not shown in the screenshots used in this paper, but can be displayed in our program.

We start by showing a naive visualization of a rule base. This visualization just shows everything contained in the rule graph as defined in section 2.2. We include this visualization technique for three reasons: firstly, it provides the initial representation of the rule base that will be the subject of the modifications described in the next sections. Secondly, by comparing the results of each transformation to this naive visualization, we can easily illustrate its effects and show its advantages. Thirdly as part of the motivation for our work, since this was our first attempt in rule base visualization whose limits made us explore other possibilities.

Figure 1 depicts the naive visualization of the example described in section 3. In the figure we have hidden the rule labels since these would clutter the picture. As you can see it is neither concise nor simple representation. In particular the number of dependencies is much too large even for this simple example, which contains just 54 rules. All in all we have 215 dependencies, most of which will never be used in actual query processing.

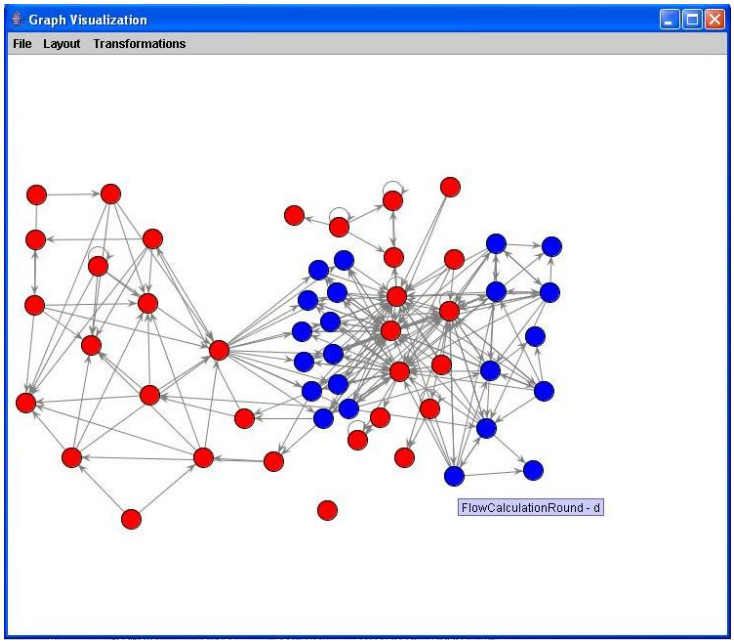

Fig. 1. *Naive Visualization of the rule base example. Blue nodes represent rules created by the user and red nodes represent axioms.*

6 Rule Base Visualization

Our approach is based on two main assumptions. Firstly that by observing the usage of a rule base we can identify which rules and which dependencies are important and should be highlighted. Secondly that it is possible to visualize the rule base at the user level, even though the rules known to the user have been transformed and extended in order for the inferences to work.

In our approach we combine three different methods. The first one utilizes usage data known for a given rule base to find and highlight frequently used rules and dependencies. The second one reconstitutes entities that have been split into many parts Finally the third one removes all axioms that have been added to the knowledge base. The second and third method also rely on the usage data in order to best perform their function. In the following we describe each of these methods in more detail.

6.1 Usage Data

As mentioned in section 2.3, the usage data consists of a number of prooftrees from queries posed to the rule base. A visualization of our example rule base enriched with this usage information is displayed in Figure 2. The rule dependencies that have actually been used in tests are displayed as bold black lines, the other dependencies are shown in grey. In addition the rule nodes are scaled depending on how often a rule has been used in a prooftree.

More formally an arc $a_r = (r_x, r_y)$ from the rule graph is displayed in bold iff there is an arg $a_p = (n_v, n_w)$ between nodes representing the firing of these rules $f_r(n_v) = r_x, f_r(n_w) = r_y$ in a prooftree

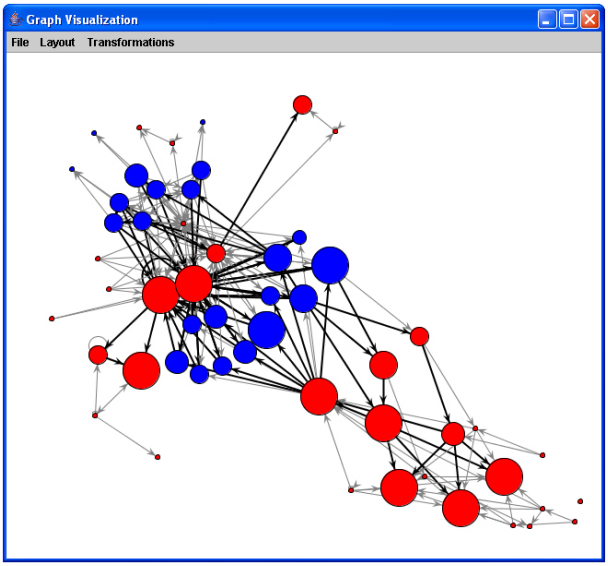

Fig. 2. *Rule graph combined with usage data.*

$G_{p,u} = (V_{p,u}, A_{p,u}), a_p \in A_{p,u}$ that is part of the usage data $G_{p,u} \in U$. The size of a rule node v_x is a function of the number of prooftree nodes $v_y \in V_{p,z}, G_{p,z} = (V_{p,z}, A_{p,z})$ that represent the firing of this rule $f_r(v_y) = v_x$. The function we use to determine the size of a vertex emphasizes the differences between rules used seldomly at the expense of usage differences between often used rules. For example a rule used once is displayed with double the diameter of a rule that was never used. The diameter of a rule used 40 times is only slightly larger than that of a rule used 30 times.

Figure 2 shows the visualization of our example rule graph and its usage data. The usage data is from 15 queries that resulted in 26 prooftrees. From the 54 rules and 215 dependencies present in the rule base, 33 rules and 83 dependencies are used in these tests and highlighted. The main advantage of this method is its ability to draw user's attention to central elements of the rule base. This way we have reached a better representation without reducing the number of rules and dependencies. This way of displaying a rule base may be applied in some cases, where displaying all rules and relationships is required - such as validation or debugging tasks.

This visualization still contains many dependencies and rules, in particular rules unknown to the user of the knowledge engineering tool. It is still hard to get a complete overview of the rule base from this picture. It gets a bit clearer when we prune everything that has not been used in any of the tests as shown in Figure 3. This way we arrive at a statistical representation of a rule base, only showing what happened during the tests. Even this representation, however, fails to capture the interaction structure of the rules created by the user. It seems to show more relations between user rules than where actually used. Consider this example: we have the user rules A,B,X and Y, there is one axiom H. Now assume that there is a number of prooftrees containing the dependencies $A \rightarrow H \rightarrow B$ (a prooftree node representing a firing of A depended on one representing a firing of H etc.) and a numer of prooftrees with $X \rightarrow H \rightarrow Y$. In the visualization we then get arcs from A and X to H and from there to B and Y - leading the user to falsely believe that A may depend on Y or X on B. Due to the very generic nature of the axiom

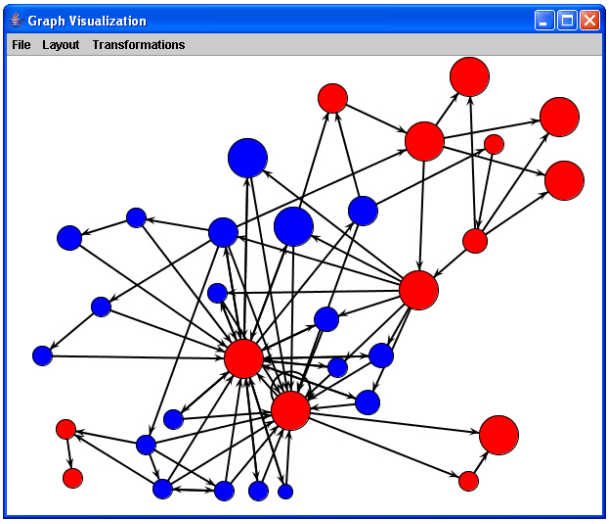

Fig. 3. *Rule graph combined with usage data, pruned.*

rules such things happen very frequently. Its evident in Figure 3 that shows two big red nodes close to the center of the picture. These vertices represent important axioms related to the class hierarchy and indirectly connect a very large number of rules.

6.2 Rejoining User Entities

Entities created by the user with the knowledge engineering tool can be translated to multiple F-logic rules in the knowledge base - we described this already in section 4. This translation process leads to a knowledge base containing more than one rule for each user entity, unnecessary cluttering the display and making it harder to see the interaction between the rules created by the user.

We deal with this challenge by rejoining all rules that where created for one user entity, the dependencies are kept during this transformation. In detail: a *user entity* can be understood as a set of rule vertices, a subset of the vertices in the rule graph. All user entities are disjunct. For each user entity we remove all its nodes from the rule graph and add one new vertice. For all edges ending in one of the removed nodes we add one that starts at the same node but ends in the new node. We do the analog for edges starting in one of the removed nodes. Edges beginning and ending in vertices of the same user entity are not included in the transformed graph. We do an analog transformation on all prooftrees, replacing nodes that represent the firing of rules that are part of a user entity. This transformation of the usage data prooftrees allows to combine the new rule graph with usage data using the techniques described in the previous section.

6.3 Hiding Axioms

As explained in section 4, we distinguish between user rules -created by the user through the rule editor- and internal rules - created automatically by the system. These internal rules or axioms are indispensable for the correct functioning of the inference engine but could confuse the user.

X

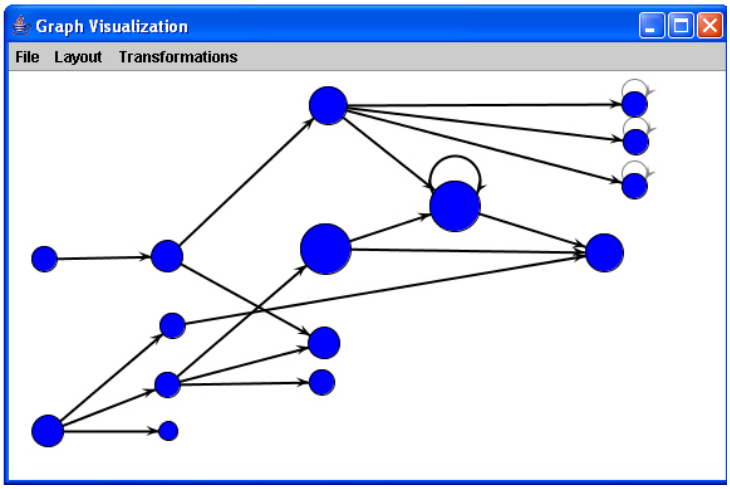

Fig. 4. *Removed Axioms: this simplified graph contains all relevant parts of the rule base.*

Hiding the axioms means that the nodes corresponding to them are deleted from the graph. The path between two user's rules traversing several axioms is replaced by a single edge between the nodes corresponding to them. This is done by an algorithm that examines the axioms one at a time. In a first steps all prooftree nodes representing the firing of this axiom are removed from the usage data prooftrees. The dependency edges are preserved, meaning that all prooftree nodes that depended on the deleted node,are connected to the nodes the removed one depended on. Assume a prooftree contains the path $X \rightarrow A \rightarrow B \rightarrow Y\ and\ Z$, where A and B are nodes representing the firing of axioms and $B \rightarrow Y\ and\ Z$ means that the firing of B depended on both Y and Z. Assuming that we process the axioms in alphabetical order this will first be transformed to $X \rightarrow B \rightarrow Y and Z$ and then to $X \rightarrow Y and Z$. We then proceed to remove all axioms and their incoming and outgoing edges from the rule graph. Finally, in the last step we add edges to the rule graph. We ensure that for each prooftree edge between nodes representing the firing of a rule A and a rule B the rule graph contains an edge between A and B.

By transforming not the rule graph directly but first the prooftree we tackle the problems described in the end of section 6.1. There we introduced two prooftree parts, $A \rightarrow H \rightarrow B$ and $X \rightarrow H \rightarrow Y$, where H was an Axiom. Had we just transformed the rule graph, deleted the axiom an reconnected the incoming and outgoing edges we would have gotton $A \rightarrow B\ and\ Y$ and $X \rightarrow B\ and\ Y$ in the rule graph. By doing the transformation with the algorithm described above we get $A \rightarrow B$ and $X \rightarrow Y$ - a better representation of the usage data. This problem is, however, only solved with respect to deleted nodes. The rule graph can still only show paths of length one, longer paths that intersect in one rule node can still not be differentiated. To tackle this challenge we plan plan to give the user the option to choose a query from the usage data and to have it highlighted as path through the rule base.

To demonstrate the effect of this method, we have hidden the axioms present in our rule base example as shown in Figure 4. Axioms - formerly represented as red nodes- are not part of the graph any more. The obtained graph contains only 15 rules and 18 dependencies, down from the original 54 rules and 215 dependencies. The interaction patterns between the user rules are now readily apparent. Even without any more information about the rule it should be possible to see the overall structure of this diagnostic

rule base: multiple "layers" of rules that integrate data about the sewers and outputs of other rules into higher level diagnosises.

After combining the three methods, we obtained a visibly better graphical representation which includes just the relevant parts of the rule base and highlights the frequently used rules and dependencies. Such representation is undoubtedly easier to manipulate and more appropriate to the user's task.

6.4 Implementation

The prototype for of this visualization system is implemented as a standalone Java application. For the actual graph we use JUNG[2] - Java Universal Network/Graph Framework. For the layout of the graphs we use the Fruchterman-Reingold algorithm[13] in an implementation that is part of the JUNG framework. Please note that Figure 4 is not directly generated through this layout algorithm. For this picture we moved a few nodes in order to better show the overall structure of the rule base. Moving of nodes is also supported by the JUNG framework.

Our program is of now not integrated with the ontology engineering tool. Currently it utilizes log files containing the prooftree forests for queries and a file representing a serialization of the rule graph. Runtime for this application has so far been unproblematic. The usage data for the example described in this paper consists of approximately 5000 prooftree nodes, around 600KB. For this example everything, i.e. starting java, starting the application, reading the files, applying the transformations, layouting and displaying the graph, is done in less than ten seconds. None of the algorithms described here is computational expensive and most could easily run in paralell, hence this kind of approach should scale easily.

7 Related Work

To our best knowledge no approach that uses usage data in the form of prooftrees for the visualization of rule bases exists. We are also not aware of any visualization approaches that address the question posed by high level knowledge engineering tools.

There exist a number of publications that address the questions of visualizing knowledge bases and rule inference mechanisms. This research field is not a modern one. It has been round since the days of early expert-systems. Graphical representation of rule-based inference has been necessary for decades to deal with the first large knowledge bases [21]. Today, several visualization approaches exist, which support the user understanding complex and formal knowledge models.

One of the most used knowledge-modeling tools is Protégé, which offers the possibility of modeling and editing Ontologies and knowledge bases. The Protégé environment may be extended by plugin components to add new functionalities [28]. Jambalaya [23] and TGVizTab [16] are two visualization plugins for Protégé, which aim at helping users to visualize Ontologies and knowledge bases by interactively navigating graphs. In these graphs, classes and instances are represented as nodes, relationships among them as edges. These visualization tools include some techniques, such as zooming, layout algorithms and edge-labeling to help the user manipulating large graphs and complex knowledge models. The tool presented in this paper differs from these tools in visualizing rules and their dependencies instead of the classes and instances.

There are some systems used to visualize Prolog clauses, like the tool for visualizing Prolog programs presented in [15]. This tool -implemented in the scope of Sotware Visualization- allows displaying conjunctions and disjunctions among predicates as well as if-then-else clauses using trees or block structures. In

[2] http://jung.sourceforge.net/

this way, it is possible to visualize rule inference and dependencies among predicates. However, reducing the complexity of the graphical representation has not been considered in this work.

Visur/Rar is a second visualization system used for visualizing Software projects in Prolog and Java. A detailed description of the system and its application in Software Visualization can be found in [9][25]. The system has been applied to knowledge bases represented in XML format. Using dependency graphs, it enables the visualization of the prooftree of a query, which formulates a specific task [27]. The authors of [27] have opted to display the rules required to answer the query rather than showing all rule dependencies. Moreover, they have focused on the use of the available XML representation and its transformation into a graphical one, diverging from our main topic, which is the visualization of rule dependencies in a clear and understandable manner.

Further visualizations of rules exist within the data mining community [6][10][17][24][26]. 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 data used for the creation of the visualization is also very different.

8 Conclusion and Future Work

We have presented the methods and their prototypical implementation for the visualization of rule bases. Results from experiments with toy examples look promising. We now plan to include our system in a knowledge engineering tool and to evaluate it in the context of a large knowledge acquisition project.

Many enhancements for the presented system are conceivable. Besides the obvious such as nicer graphics, zooming capabilities or better integration with the ontology engineering tool these are in particular: the inclusion of facts, integrated visualization of inference processes and support for fault localization. It is obvious that it would be desirable to have a visualization that includes both the facts and the rules from the knowledge base, albeit it is less clear how that would look like. The rule graph could also be adapted to show the evaluation of a query as a path through this graph - either by animating it or by highlighting the rules and dependencies that are involved. Such visualization of the inference process could further aid the user in understanding a rule base. Finally the system could work as a visual aid for fault localization by overlaying the paths through the rule graph taken by failed and successful test queries.

References

1. J. Angele. How to write f-logic programs. Technical report, Ontoprise GmbH, 2005.
2. Jürgen Angele, Harold Boley, Jos de Bruijn, Dieter Fensel, Michael Kifer Pascal Hitzler, Reto Krummenacher, Holger Lausen, Axel Polleres, and Rudi Studer. Web rule language (wrl). Technical report, University of Karlsruhe, 2005.
3. Jürgen Angele and Georg Lausen. Ontologies in f-logic. In *Handbook on Ontologies*, pages 29–50. Springer, 2004.
4. S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. Swsl-rules: A rule language for the semantic web. Technical report, Semantic Web Services Language Committee of the Semantic Web Services Initiative, 2005.

5. Harold Boley. Knowledge bases in the world wide web: A challenge for logic programming. In Koen De Bosschere Paul Tarau, Andrew Davison and Manuel Hermenegildo, editors, *Proceedings JICSLP'96 Post- Conference Workshop on Logic Programming Tools for INTERNET Applications*, 1996.
6. 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.
7. Stefan Decker, Dan Brickley, Janne Saarela, and Jürgen Angele. A query service for rdf. In *QL*, 1998.
8. Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In *DS-8*, pages 351–369, 1999.
9. D.Seipel, M.Hopfner, and B.Heumesser. Analyzing and visualizing prolog programs based on xml representations. In *Proceedings of the 13th International Workshop on Logic*, 2003.
10. Usama Fayyad, Georges Grinstein, and Andreas Wierse. *Information Visualization in Data Mining and Knowledge Discovery*. Morgan Kaufmann, 2001.
11. Dieter Fensel, Stefan Decker, Michael Erdmann, and Rudi Studer. Ontobroker: The very high idea. In *FLAIRS Conference*, pages 131–135, 1998.
12. 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.
13. Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
14. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.
15. C.A.McK. Grant. *Software Visualization in Prolog*. PhD thesis, University of Zurich, 2001.
16. H.Alani. Tgviztab: An ontology visualisation extension for protégé. In *In Proceedings of Knowledge Capture (K-Cap'03), Workshop on Visualization Information in Knowledge Engineering*, Sanibel Island, Florida, USA, 2003.
17. Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 2000.
18. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. Swrl: A semantic web rule language combining owl and ruleml. Technical report, W3C Member submission 21 may 2004, 2004.
19. Michael Kifer, Jos de Bruijn, Harold Boley, and Dieter Fensel. A realistic architecture for the semantic web. In *RuleML*, pages 17–29, 2005.
20. Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.
21. 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.
22. Massimo Marchiori and Janne Saarela. Query + metadata + logic = metalog. In *QL*, 1998.
23. M.A.Storey, M.Musen, J.Silva, C.Best, and N.Ernst. Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in protégé. In *Workshop on Interactive Tools for Knowledge Capture*, 2001.
24. M.Burch, S.Diehl, and P.Weißgerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 37 – 46, 2005.
25. M.Hopfner, D.Seipel, and J.W.von Gudenberg. Comprehending and visualizing software based on xml–representations and call graphs. In *Program Comprehension,11th IEEE International Workshop on Logic*, pages 290– 291, Tubingen, Germany, 2003.
26. 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, Germany, 2002.
27. Dietmar Seipel, Joachim Baumeister, and Marbod Hopfner. Declarative querying and visualizing knowledge bases in xml. In *15th International Conference of Declarative Programming and Knowledge Management (INAP-2004)*, pages 140–151, 2004.

28. M.A Storey, R.Lintern, N.Ernst, and D.Perrin. Visualization and protege. In *Proceedings of 7th International Protege Conference*, 2004.
29. Guizhen Yang and Michael Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *CoopIS/DOA/ODBASE*, pages 1013–1032, 2002.