Understanding Rulelog Computations in Silk

Carl Andersen*, Brett Benyo*, Miguel Calejo**, Mike Dean*, Paul Fodor***, Benjamin N. Grosof†, Michael Kifer***, Senlin Liang***, and Terrance Swift‡

Abstract. Rulelog is a knowledge representation and reasoning language based on logic programming under the well-founded semantics. It is an extension of the language of Flora-2 and so supports inheritance and other object-oriented features, as well as the higher-order syntax of Hilog. However, Rulelog rules may also contain quantifiers and may be contra-positional. In addition, these rules are evaluated in the presence of defeasibility mechanisms that include rule cancellation, rule priorities, and other aspects. Rulelog programs are sometimes developed by loosely coordinated teams of knowledge engineers (KEs) who are not necessarily programmers. This requires not only declarative debugging support, but also support for profiling to help KEs understand the overall structure of a computation, including its termination properties. The design of debugging and profiling tools is made more challenging because Rulelog programs undergo a series of transformations into normal programs, so that there is a cognitive distance between how rules are specified and how they are executed.

In this paper, we describe the debugging and profiling environment for Rulelog implemented in the integrated development environment of the Silk system. Our approach includes an interface to justification graphs, which treat why-not and defeasibility as well as provenance of the rules supporting answers. It also includes tools for trace-based analysis of computations to permit understanding of erroneous non-termination and of general performance issues. For semantically correct cases of the non-terminating behavior, Silk offers a different approach, which addresses the problem in a formally sound manner by leveraging a form of bounded rationality called restraint.

1 Introduction

Knowledge Representation and Reasoning (KRR) languages are often expressive fragments of first-order logic, such as OWL-DL [10], or are fixed point logics based on ASP [2] and its extensions. While such languages offer concise representations and use powerful reasoning techniques, they often are not scalable

* Raytheon BBN Technologies, USA
** Declarativa, Portugal
*** Stony Brook University, USA
† Benjamin Grosof & Associates, LLC, USA
‡ CENTRIA, Universidade Nova de Lisboa, Portugal
enough to address applications in domains such as the semantic web. As an alternative, KRR languages and systems have also been developed whose inferencing is of lower computational complexity, such as the description logic $\text{EL}^+$ [1], or the rule based systems Flora-2 [17] and Silk.

The Silk system is based on a language called Rulelog, which combines a number of features from logic programming (LP). Silk has a 3-layer architecture. The top (Java) layer provides much of the connectivity to other systems as well as some of the transformations, such as the omni transform discussed later. The second layer, Flora-2, supplies most of the functionality of Rulelog. The two upper layers compile Rulelog programs into normal programs (containing recursion and logical functions) and execute them under the well-founded semantics using the XSB system [15] 1. The main features of Rulelog add considerable expressivity over normal logic programs. As with XSB, answers may have an truth value of undefined due to a form of bounded rationality called radial restraint [5], in addition to being undefined due to a loop through negation. As with Flora-2, the frame-based syntax is supported, heavy use can be made of Hilog, and rules may be defeasible. In addition, Rulelog allows use of a general first-order syntax not only in rule bodies as in [6], but also in rule heads giving rise to omni rules. The use of these features raises a number of issues both in debugging and in understanding the behavior of Rulelog derivations.

Traditionally, the justification problem is the problem of explaining missing or unexpected answer [11, 12]. Justification is made more complex by several features of Rulelog. First, Rulelog inferencing is based on the well-founded semantics, leading to the task of explaining answers whose truth value is undefined. In addition, the use of defeasible reasoning and of Hilog can lead to unexpected inferences. Furthermore, transformations are used to implement omni rules, Hilog, and defeasibility. Together, these transformations can make the compiled rules look substantially different than the source.

The performance/termination problem is the problem of indicating why a derivation has taken up more resources than expected — including non-termination as an extreme case. Addressing this problem has been especially important for Rulelog users. As will be shown below, new and sometimes cyclic dependencies arise from the use of omni rules, from defeasible reasoning, and from the use of Hilog. While these cyclic dependencies can be addressed by tabling, unexpected cyclic dependencies can lead to huge mutually recursive components. In addition, as logical functions are allowed, unexpected cycles can lead to non-terminating queries. These problems with unexpected dependencies are exacerbated by the fact that Rulelog is aimed at knowledge engineers (KEs) who are competent in logic, but who are not necessarily computer programmers. These KEs share a common background vocabulary in developing inter-dependent rules, but are often loosely coordinated.

This paper discusses the approaches to the justification and performance problems that are part of the Silk system. Section 2 discusses the Rulelog lan-

1 Rulelog has also been partially implemented using the Cyc reasoner of Cyicorp.
guage and how it has been implemented via Flora-2 and XSB. Section 3 discusses the approaches to the justification problem, while Section 4 discusses how performance is assessed via traces of Flora-2 and XSB query evaluations. We note that many of these approaches are still under development, and in each section we note the current status of each approach.

2 Rulelog and its Implementation

Rulelog, as implemented in the Silk system, supports not only direct construction of knowledge bases, but also interchange with knowledge formatted in various semantic standards such as OWL-RL and RDF. Rulelog also provides a good target for text-based authoring of knowledge [3], because of its ability to express defeasible formulas as axioms. In this section we present the main semantic and syntactic features of Rulelog that affect debugging and profiling, and then briefly discuss its implementation in the Silk system.

2.1 Omni Rules and Hilog

In this paper, we make use of general terminology of logic programs, but adopt some syntax of Rulelog. In Rulelog, variables are designated by tokens that begin with ’?’; a literal is an atom $A$ or its explicit negation $\text{neg } A$; a default literal is either an objective literal $O$ or its default negation $\text{naf } O$. A normal rule is designated as $\text{Head} : - \text{L}_1 \text{ and } \ldots \text{ and } \text{L}_n$, where $\text{Head}$ is an atom and each $\text{L}_i$ is a default literal. Bodies of normal rules can be made to have a more expressive syntactic form using the well-known Lloyd-Topor (LT) transformation[6]. For example, the rule

$$p_{\text{equivalent}}(\forall X,?Y) : - \forall Z (p(Z,X) <=> p(Z,Y)).$$

is LT-transformed into a sequence of normal rules.

Omni rules extend the LT-transformation by allowing first-order formulas to occur in rule heads as well as rule bodies. To take an example, a statement of molecular biology “A contractile vacuole is inactive in an isotonic environment” can be modeled using the omni rule

$$\forall x (\text{contractile(vacuole})(x)) => \forall x (\text{isotonic(environment)}(x)) => \text{inactive(in(x)})),$$

which is transformed into the rules

$$\neg \text{isotonic(environment)(x)} : - \text{contractile(vacuole)(x)} \text{ and } \neg \text{inactive(in(x))},$$

$$\neg \text{contractile(vacuole)(x)} : -$$

2 In the literature, literals in our terminology are sometimes called objective literals, and default literals are called literals.
isotonic(environment)(?x9) and neg inactive(in(?x9))(?x6).
inactive(in(?x9))(?x6) :-
contractile(vacuole)(?x6) and isotonic(environment)(?x9).

As can be seen from this example, the semantics of omni rules is entirely transformational. Note that each of these rules makes use of Hilog, which allows any functor of a term to be a variable or a compound term, rather than simply an atom. Flora-2 itself allows explicit negation, and applies another transformation to remove the explicit negation, resulting in normal rules executed by XSB.

2.2 Rule Descriptors

While Prolog or ASP programmers typically think in terms of predicates, KEs often think in terms of rules when they want to understand inferences or make declarations about overriding. Accordingly, Rulelog supports meta-information about rules through descriptors, which rely on the frame syntax of Flora-2 and may themselves be defined as rules. As an example, consider a rule indicating that a eukaryotic cell has a visible nucleus as in the following code.

```flora-2
@!{r1[tag->eukc, kgroup->g1]}
eukaryotic(cell)(?x1) :- has(?x1,?x2) and visible(nucleus)(?x2).

@@strict ?x[owned->Benj] :- @!(?x[kgroup ->g1]).
```

The above syntax means that the rule about eukaryotic cells has the unique identifier r1; the rule also has a tag attribute whose value is eukc (rule tags are used by argumentation theories to determine whether the rule can be opposed or overridden). The second rule has a descriptor @@strict, which means that the rule is not defeasible. The rule itself says that any rule whose descriptor meta-information says that it belongs to the kgroup g1 is owned by Benj.

2.3 Argumentation Theories

As a KRR language, Silk makes heavy use of argumentation theories [16] that affect the derivations made by rules. Defeasibility is specified via two user-defined predicates: opposes/2 and overrides/2. Two atomic formulas oppose each other if no model of a program may contain both atoms: e.g., an atom and its explicit negation oppose each other, but opposition can capture other types of contradictions. In addition, rules can be prioritized. Each rule has an explicit or implicit tag. Implicit tags default to rule ids, but tags generally are distinct from rule ids: tags are used for prioritizing rules, so different rules may have the same tag. Given two rules, one may override the other and so be given preference, which is written as override(tag1,tag2), where tag1 and tag2 are the tags of the respective rules. Figure 1 shows a highly simplified argumentation theory. Such a theory is applied to a program by a transformation as follows [16]. Each clause @!{id[tag->t]} H :- B whose head is a defeasible predicate is rewritten as
H :- B, naf defeated(t); clauses for non-defeasible rules (called strict) are not altered. For atoms \( A_1 \) and \( A_2 \), if \( A_1 \) and \( A_2 \) are both derivable and oppose each other but neither overrides the other, \( A_1 \) and \( A_2 \) mutually rebut each other. If, in addition, \( A_1 \) overrides \( A_2 \), we say that \( A_1 \) refutes \( A_2 \). Within Silk, the compilation of a argumentation theory ensures that rebutted atoms have an undefined truth value, as do atoms that refute themselves (i.e. if the overrides/2 predicate is cyclic).

defeated(?T) :- (refutes(?T2,?T) or rebuts(?T2,?T)), candidate(?T2).
refutes(?T,?T2) :- conflicts(?T,?T2), overrides(?T,?T2).
rebuts(?T,?T2) :- conflicts(?T,?T2), naf overrides(?T,?T2).
conflicts(?T,?T2) :- (opposes(?T,?T2) or opposes(?T2,?T)), candidate(?T2).
candidate(?T) :- headof(?T,?H), ?H.

Fig. 1. A Highly Simplified Defeasibility Theory

Argumentation theories used in practice are far more complex than that of Figure 1. A primary motivation is to capture sophisticated arguments a human reasoner might make in applying default reasoning to a problem. In justifying Rulelog inferences, it is important to communicate to the user the steps when a truth value is concluded due to the use of defeasibility.

2.4 Use of Three-Valued Logic

The well-founded semantics assigns a partial model to a program \( P \), where the truth value of certain atoms in the ground instantiation of \( P \) may be neither true nor false, but undefined. This third truth value was added to handle situations in which an atom \( A \) had no true derivations; but where \( A \) had at least one derivation which depended on naf \( A \), a situation we term a negative loop.

In systems like XSB and Flora-2, which support the well-founded semantics, the third truth value can be put to use for other reasons as well. For instance, in ISO Prolog exception (or error) conditions cause a computation to abort if the error is not specifically caught. However, this procedural approach to handling errors could be replaced by a semantics that assigns an atom \( A \) a truth value of undefined when \( A \) has no true derivations, but \( A \) has at least one derivation that depends on an exception. This approach properly generalizes the well-founded semantics from negative loops to other classes of exceptions. As noted in [9], a three-valued approach can be superior for debugging to the ISO Prolog approach, as the entire search space for proving an atom \( A \) can be examined, including possible true derivations of \( A \) that might not be seen if an exception were thrown.

Restraint The third truth value may be generalized yet again, leading to a type of bounded rationality termed restraint. Consider the program:
\begin{verbatim}
p(0). p(s(X)) :- p(X). p(X) :- p(s(X)).
\end{verbatim}

If the goal \texttt{?- p(a)} is given to this program, then a tabled evaluation may create an infinite number of subgoals: \texttt{p(a)}, \texttt{p(s(a))}, \texttt{p(s(s(a)))}, and so on. In addition, the goal \texttt{p(Y)} to the above program has an infinite number of answers.

Each of these situations can each be handled by a different tabling technique. If \textit{subgoal abstraction} is used and if a program is \textit{safe} (as is the preceding program), then it can be ensured that only a finite number of subgoals are created [13].\footnote{Note that the tabling technique of call subsumption will not help as none of the above subgoals subsume one another.} If the technique of \textit{radial restraint} [5] is used, it can be ensured that only a finite number of answers are generated. To see how this works, suppose that the following declaration were made for \texttt{p/1}:

\texttt{\textit{:- table p/1 as subgoal\_abstract(2), answer\_abstract(3).}}

Such a specification causes abstraction in two cases. Any subterm in a subgoal for this predicate whose depth is greater than 2 is abstracted (replaced with a variable), so that the only subgoals created for \texttt{?- p(a)} would be \texttt{p(a)} and \texttt{p(s(X))}. Similarly, any subterm in an answer for this predicate whose depth is greater than 3 would also be abstracted \textit{and assigned the truth value undefined}. Thus, the goal \texttt{?- p(Y)} would return the answers \texttt{p(0)}, \texttt{p(s(0))} and \texttt{p(s(s(0)))}; but \texttt{p((s(s(s(0))))), p((s(s(s(s(0)))))}, etc. would be abstracted to \texttt{p((s(s(s(X))))}} (and assigned the truth value of \textit{undefined}. Because radial restraint uses the truth value \textit{undefined} to answers when abstracting, it preserves the soundness of derivations, even in the presence of negation.

As a form of \textit{unsafety restraint}, the XSB predicate \texttt{unot/1} is a type of tabled negation that makes the default literal \texttt{unot(A)} undefined if \texttt{A} is non-ground. This can be seen as a type of restraint, as in principle constructive negation could be used, though constructive negation might raise the computational complexity of a derivation, or make a derivation non-terminating. In Flora-2, the use of \texttt{unot/1} is combined with the ability to delay evaluation of non-ground negative subgoals through the use of the \textit{when} library. The truth value of a non-ground negative subgoal is made \textit{undefined} only when it is determined that variables in the subgoal can never be instantiated, so that further delaying is futile.

Using \textit{rule-based} restraint, general forms of bounded rationality can be programmed explicitly. Conceptually, rule-based restraint is invoked using rules for \textit{skipping} similar to those for \textit{opposes} and \textit{overrides} (Section 2.3). Skipping is easiest understand by example. Consider the following fact and a rule:

\begin{verbatim}
step(1). step(N\_out) :- step(N), N\_out is N + 1.
\end{verbatim}

which might be used, e.g., to define a planning horizon. Note that radial restraint could not be used to soundly restrict the planning horizon generated by \texttt{step/1} as integer arithmetic does not use the term structure needed by that technique. However the rule:
would cause the body of the second rule for \textit{step/1} to be compiled to

\[
\text{step}(\text{N}out;\neg \text{step}(\text{N}), \text{N}1 \text{ is } \text{N} + 1, \text{skip}(\text{step}(\text{N}1)) \rightarrow \text{N}out = \text{N}1 ; \text{undefined}).
\]

If the body of the skip statement is false (\(\text{N} \leq 10\)), no rewrite is made to the variable, \text{N}out of \textit{step/1}. Otherwise \text{N}out is abstracted to an anonymous variable, and the goal \textit{undefined} is called, so that \textit{step(N)} is \textit{undefined} when \(\text{N} > 10\). Thus the goal \textit{step(X)} would have 11 answers, where the first 10 bind \text{X} to the integers 1 to 10 with truth value \textit{true}, while the eleventh does not bind \text{X} and has the truth value \textit{undefined}.

Note that a similar skipping rule could be used to program both radial and unsafety restraint, so that rule-based restraint is more general than either of these approaches. However, both radial restraint and unsafety restraint are implemented in the XSB engine, so are more efficient than rule-based restraint. Rule-based restraint can be mixed into an argumentation theory, giving rise to \textit{restrained argumentation theories}.

### 3 Justification

\textit{Justification}, a method of explaining how a derivation was made, has a long history in KRR, starting with \textit{truth maintenance systems} (e.g., [8]) and applied in LP to tabled logic programming [11] and ASP [12]. Justification for Rulelog requires addressing the various features described in Section 2. For instance, recall that, in defeasible reasoning, a literal might be \textit{false} or \textit{undefined} because it is derived by rules that are \textit{defeated} by other rules. In those cases, it is necessary to explain how and why those rules were defeated, and whether prioritization was involved. A key aspect is not only to explain why literals are \textit{true} or rules are active, but why literals are \textit{false} (or \textit{undefined}), and rules are defeated. In addition, if a literal has a truth value \textit{undefined}, the user should be able to understand whether this is due to a negative loop, to restraint, to unsound negation, or some other error condition—or due to some combination of these reasons. In addition, if the inheritance mechanisms of Flora-2 are used, an explanation must be given to why a given predicate was chosen to derive an attribute value for an object.

Within Silk, justification is visualized graphically, through the Silklipse environment ([4] described an early version). In addition, Silk supports rule-based transformation of the justification information: allowing literals to be displayed as English text, permitting sets of literals to be summarized or omitted from the justification, and so on. Figure 2 shows a screenshot of a navigable justification in Silklipse for why the statement “\textit{cell53 has a nucleus}” is (default) \textit{false}. Justification rules have transformed some lines into English text; for instance, the first line reads: “\textit{It is not the case that cell53 has a nucleus}.”. Other
lines appear as Rulelog literals, such as the forth line: \texttt{cell52 \# red(blood(cell))}, where “\#” means “is an instance of the class”. Each line is associated with one or more icons that indicate how a literal has been used in a derivation. The icon “G” indicates a literal (perhaps translated into English) that is a (sub)goal. “A” indicates an argument for support of the literal—in the sense of prioritized argumentation in defeasibility. Operationally an argument may be a rule body, but can also include literals from the head of a contra-positional omni rule. “F” indicates a fact, i.e., a literal that was directly asserted, while “P” indicates prioritization meta-information, i.e., that one rule’s tag overrides another tag. The color green indicates that a literal is true, while red indicates that a literal is false. Similarly, a green bang (“!”) indicates a undefeated argument, while a red down arrow (“↓”) indicates an argument that has been defeated—perhaps refuted by conflicting arguments with higher priority. The plus sign (“+”) just to the right of “G” indicates that there are more arguments to see. When the “+” is black it indicates there are both pro and con arguments to see; when green, it indicates there are more pro arguments but not more con arguments to see. Finally, a black bar (“—”) indicates an argument for the neg (strong negation) of the goal literal.

In this example, the relevant asserted logical rules in the KB can be described in English as follows:

\begin{quote}
\texttt{cell52 is a red blood cell.}
\texttt{Eukaryotic cells have nuclei. (This rule has tag r1.)}
\texttt{Red blood cells are a subclass of eukaryotic cells.}
\texttt{Red blood cells do not have nuclei. (This rule has tag r2.)}
\texttt{r2 has higher priority than r1.}
\end{quote}

Silk’s justification system has been heavily used by KEs, so that obtaining good performance for the justification has been a critical task. The first approach was to make use of a justification transformation of the rules, in a spirit similar to [11]. This approach quickly proved unwieldy due to the transformations needed for defeasibility and for omni rules. For instance, the omni rule in Section 2.1 generated 98 justification rules. As an alternative, Silk now invokes a meta-interpreter to construct a justification graph when a user requests justification. The heavy use of tabling in Flora-2 means that the justification graph can be reconstructed quickly in most cases by making use of information in the tables themselves.

\textit{Status of Justification} The current version of justification in Silk was tested on a previous generation of argumentation theories, and does not yet fully support the deeper argument refutations used in ATCO (Section 2.3), or the various types of restraint discussed in Section 2.4. However, even with these limitations, it has been actively used by KEs.
4 Performance Profiling through Trace-based Analysis

Justification as previously described addresses why a literal is *true, false* or *undefined*, and so relies on the semantics of Rulelog. Accordingly, the justification graph itself is largely independent both of the transformations of Rulelog into Flora-2 code and then into normal programs, and of the tabled evaluation used by XSB. Understanding the performance profile of derivation (and to some extent whether it terminates) depends on the characteristics of how a derivation was actually implemented. Performance bottlenecks will differ depending on whether a derivation is based on tabling or on bottom-up evaluation; on whether the tabling uses call-variance or call subsumption [15], and so on. The tools presented here for trace-based analysis rely on characteristics of XSB's tabled evaluation, but the user-level tools try to minimize the amount of information a user needs to know about the particulars of tabling.
4.1 Table Dump: Examining Subqueries, Answers, and Rules

The most direct way to understand the performance of a tabled computation is to look at the tables themselves. Table Dump is a tool that does just that. A user enters a term $T$, and Table Dump returns information about all tables whose subgoals are subsumed by $T$. If $T$ is variable, information about all tables is returned. As Silk computations may produce hundreds of thousands of tables, information can be returned in a summary form, and users may “drill-down” by querying successively more instantiated terms. Figure 3 shows a screenshot of a navigable view of table dump information in Silklipse for the schematic Rulelog term $?A[?B->?C]$ — a frame syntax for asking about attribute-value pairs $?B->?C$ for some object $?A$. Information is given about the answers to a subgoal, the number of times it has been called, and even the rule (or rules) used to call the subgoal. Table Dump thus helps a KE to identify bottlenecks in the knowledge base and then take measures such as adding appropriate guards to rules or reordering subgoals within rules.

The Table Dump tool for Silk is based in part on the table_dump family of predicates in XSB. XSB’s table_dump is implemented using XSB’s table inspection predicates, and relies on the XSB engine’s maintenance of a count of the number of times each table is called. However XSB’s tables maintain information about subgoals, and does not provide the rule-specific information seen in Figure 3. To support this, Flora-2 performs a transformation that embeds rule
ids in each predicate of the rules in question. These predicates are tabled so that when information is returned from XSB’s `table_dump`, information about the rules is extracted and shown in Silklipse.

4.2 Trace-Based Analysis Based on Forest Logging

Although simple and powerful, the table dump approach lacks two main features needed to fully address the performance/termination problem. First, it does not provide an overview of how given subqueries in a derivation relate to one another through rules, and does not display information about the recursive components whose computation is central to a Rulelog derivation. Second, no information is provided about the order of events in a derivation, such as when subqueries were made, answers derived, and so on.

Within Silk, details of a Rulelog derivation can be reconstructed through another kind of trace-based analysis. XSB provides a mechanism to create a more dynamic trace or log of a derivation, called a forest log [14]. Using such a log, the structure of even very large recursive components can be analyzed, and non-terminating derivations detected. This subsection first overviews forest logs and afterwards discusses the analysis routines based on the logs.

**Forest Logging** The form of tabling used by XSB is called SLG resolution. The operational semantics of SLG evaluation (and hence a Rulelog derivation) can be modeled as a sequence of forests of trees, where each tree corresponds to a tabled subquery $S$ and represents the immediate subqueries that $S$ produces along with any answers to $S$. In fact, each SLG operation is modeled as a function from forests to forests that creates a new tree, or adds a node or label to an existing tree.

Within XSB, SLG resolution is executed using a byte-code virtual machine analogous to that used by Java. An internal XSB flag can be set so that any byte-code instruction that corresponds to a tabling operation will log information about itself and its operands as a Prolog-readable term. For instance, if (tabled) subgoal $S_1$ is called in the context of subgoal $S_2$, and it is the first time $S_1$ is called in an evaluation, a fact of the form

```
table_call(S_1, S_2, new, ctr)
```

is logged, where $ctr$ (mnemonic for “counter”) is a sequence number for the fact. When a derivation ends or is interrupted, the log can be loaded into XSB and analyzed as a set of Prolog facts. Within XSB, the logging system is written at a very low level for efficiency. Turning on full logging usually does not slow down performance by more than 70-80%. XSB also provides routines to load logs and index their facts on various arguments. Based on the logging libraries, logs containing hundreds of millions of facts have been loaded and analyzed [14].
Analyzing Recursive Components  Once a log has been loaded, a user may ask for an overview of a computation, which provides information on the total number of calls to tabled subgoals, the number of distinct tabled subgoals, the number of answers, and so on. In addition, the overview provides aggregate information on the number of mutually recursive components, and the number of subgoals in the components. Finally, the overview contains information indicating how stratified the negation was in a derivation by listing the total number of atoms whose truth value was undefined, along with a count of the various SLG operations used to evaluate well-founded negation.

Some derivations may give rise to very large recursive components—due to an unanticipated effects of the Hilog, omni, and defeasibility transformations; to a knowledge base that is not sufficiently modularized, or to other reasons. The analysis routines allow a given recursive component to be examined, by listing the subqueries in the component, along with the pairs of calling and called subgoals within the component.

By examining this output, users can usually fix whatever problems gave rise to large recursive components. However for a very large component $\mathcal{C}$, the number of subqueries in $\mathcal{C}$ may be on the order of $10^5$ or more and the number of calling/caller pairs may be on the order of $10^6$. In such a case, displaying every subquery or pair may be confusing at best. The analysis routines thus provide several abstraction routines that allow a user to coalesce similar atoms. For instance, if a component contained the subqueries $p(a,X)$, $p(b,X)$, $p(c,X)$ ..., the analysis routines could use mode abstraction to coalesce all of these terms to $p(bound,free)$, or even predicate abstraction to coalesce all these terms to $p/2$.

Recursive component analysis together with abstraction of atoms has been used to analyze the behavior of reasoning that was translated from Cycorp’s inference engine into the Silk implementation of Rulelog, among other applications.

Analyzing Runaway: Terminyzer  Runaway computation occurs when a query does not terminate or takes too long to come back with an answer. The first type of problem occurs typically due to the presence of function symbols and the second is largely due to computations that produce very large intermediate results most of which could be avoided with smarter evaluation strategies, such as subgoal reordering.

One sophisticated diagnostic tool we have developed to tackle the non-termination problem is called the Terminyzer (short for “(non-)Termination Analyzer”) [7]. This tool relies on the previously described forest logging mechanism, which records the various tabling events that occur in the underlying inference engine XSB [15]. Among others, forest logging records when the different subgoals are called and when they receive answers. Terminyzer performs different kinds of analysis, such as call-sequence analysis and answer-flow analysis, and identifies the sequences of subgoals and rules that are being repeatedly called and in this way cause non-terminating computation.
Terminyzer also has a heuristic that can suggest to the user that the system be allowed to reorder subgoals at run time and thus avoid non-termination. For instance, in a composite subgoal \( p(?X,?Y), q(?X) \) Terminyzer may detect that \( p(?X,?Y) \) is an infinite predicate. However, this infinity may be due to the infinite number of \(?X\)-values. If \( q(?X) \) is finite and is evaluated first, non-termination will not occur. In such a case, Terminyzer may suggest the user to wrap the offending subgoal with a suitable delay quantifier—a novel facility supported by Flora-2 and Silk. For instance, if the above subgoal is rewritten as \( \text{wish(ground(?X))} \cdot p(?X,?Y), q(?X) \), the system will not try to evaluate \( p(?X,?Y) \) unless \(?X\) is bound. If it is not bound, the evaluation of \( p(?X,?Y) \) is postponed and \( q(?X) \) will be evaluated next. If this binds \(?X\) then all is well and \( p(?X,?Y) \) can be evaluated next without a runaway. If \(?X\) is still unbound, some other subgoal may possibly bind it, so \( p(?X,?Y) \) remains delayed. Only when the system determines that \(?X\) cannot be bound no matter what, is \( p(?X,?Y) \) submitted for evaluation. If this happens, the user would have to use the information provided by Terminyzer to decide whether the runaway is a mistake or is semantically justified. In the first case, this information will help the user fix the mistake; in the second, restraint (discussed in Section 2.4) could be used to prevent the runaway.

### Invoking Trace-Based Analysis

The interfaces to trace-based analysis are based on an XSB meta-predicate, called `timed_call(Goal,Handler,Interval)`, which calls `Goal` and interrupts the computation of `Goal` every `Interval` milliseconds to call `Handler`. When `Handler` terminates, `Goal` is resumed. Silk’s use of Interprolog allows XSB and Java processes to call each other recursively. Thus in Silklipse, the graphical interface to Silk, `Handler` is a call to an interrupt window that allows access to Table Dump, forest logging, Terminyzer, and other routines. Given this facility a user may specify that a computation that is expected to be long running be interrupted every \( N \) seconds. At interrupt time, the user may explore the current tables in the computation, turn forest logging on or off, and perform termination analysis. Depending on the information given, the user may then either continue or abort the computation.

#### Status of Trace-Based Analysis

The Table Dump tool and forest logging are both fully integrated into the Silklipse environment and interrupt mechanism, as are hooks for Terminyzer (Figure 4 shows an example). The tools in Section 4.2 for analyzing recursive components are not yet available from Silklipse.

### 5 Discussion

As the Rulelog engine was implemented in Flora-2 and XSB, features of these systems sparked further development of Rulelog, in particular the development of restraint. Typically, the development of debugging and profiling facilities and their inclusion into Silklipse has trailed engine development. Accordingly Silk’s
justification system does not yet fully support the features of ATCO or restraint and analysis of mutually recursive components is not yet implemented within the Silklipse interrupt screen. All in all, the development of Rulelog and Silk have spurred research into extensions of previously known debugging techniques, such as justification. They have also fostered the new trace-based analysis techniques of forest logging and the Terminyzer, along with ongoing research into better explanations of the use of restraint within computations.

Funding for Silk from Vulcan Inc. terminated in April 2013. Vulcan agreed to allow the Flora-2 code to become open-source, so that all of the Rulelog engine, with the exception of omni rules, is now fully open-source.\footnote{All Vulcan-funded work on XSB has been open-source.} Although Vulcan retains rights to the Silklipse code, a new development environment for Rulelog, called \textit{Fidji} is under active development by the authors of this paper, with the goal of a prototype release by the end of 2013.

\textbf{Acknowledgements} This work was supported by Vulcan, Inc., as part of the Halo Advanced Research project. Michael Kifer and Senlin Liang were also partly supported by NSF grant 0964196. Thanks to Paul Haley (Automata, Inc.) and Keith Goolsbey (Cycorp) for helpful discussions, to Walter Wilson who implemented \texttt{table dump}, and FCT Project ERRO PTDC/EIACCO/121823/2010.
We also thank the anonymous referees for suggesting the many improvements to an earlier draft.

References