A Practical Analysis of Non-Termination in Large Logic Programs

Senlin Liang and Michael Kifer

Department of Computer Science
Stony Brook University
Stony Brook, NY 11794, USA
(e-mail: {sliang, kifer}@cs.stonybrook.edu)

submitted 10 April 2013; revised 23 May 2013; accepted 23 June 2013

Abstract

A large body of work has been dedicated to termination analysis of logic programs but until recently little has been done to analyze non-termination. In our opinion, explaining non-termination is a much more important task because it can dramatically improve a user’s ability to effectively debug large, complex logic programs without having to abide by punishing syntactic restrictions. Non-termination analysis examines program execution history when the program is suspected to not terminate and informs the programmer about the exact reasons for this behavior. In (Liang and Kifer 2013), we studied the problem of non-termination in tabled logic engines with subgoal abstraction, such as XSB, and proposed a suite of algorithms for termination Analyzer, called Terminyzer. These algorithms analyze forest logging traces and output sequences of tabled subgoal calls that are the likely causes of non-terminating cycles. However, this feedback was hard to use in practice: the same subgoal could occur in multiple rule heads and in even more places in rule bodies, so Terminyzer left too much tedious work for the user to do manually.

In this paper we propose a new suite of algorithms, Terminyzer+, which closes the usability gap or, at least, narrows it down drastically. Terminyzer+ can detect not only sequences of subgoals that cause non-termination, but, importantly, the exact rules where they occur and the rule sequences that get fired in a cyclic manner, thus causing non-termination. This makes Terminyzer+ suitable as a back-end for user-friendly graphical interfaces on top of Terminyzer+, which can greatly simplify the debugging process. Such an interface is currently under construction in the SILK project.¹

In addition, we make a step towards automatic remediation of non-termination programs by proposing an algorithm that heuristically fixes some causes of such behavior. Furthermore, unlike Terminyzer, Terminyzer+ does not require the underlying logic engine to support subgoal abstraction, although it can make use of it.

Terminyzer+ has been implemented for FLORA-2, and here we report experimental studies, which confirm its effectiveness. All tests used in this paper are available online.²

KEYWORDS: terminyzer, non-termination analysis, termination analysis, rule id, logic programming, forest logging, tabling, subgoal abstraction.

¹ silk.semwebcentral.org
² www.cs.sunysb.edu/~sliang
1 Introduction

The problem of run-away computations in logic programs is much more serious than in procedural programming because of the declarative nature of the logic languages and the large gap between the declarative semantics and the actual evaluation strategy. This problem is even more vexing in high-level logic languages such as Flora-2 and SILK, which position themselves as tools for developing complex knowledge bases by knowledge engineers who are not programmers. This type of users cannot be expected to debug the procedural aspects of the rule bases that they create and thus they require special support. Non-termination has been flagged as a key issue standing on the way of creation of complex biological knowledge base in SILK project, where the use of function symbols is more common due to higher-order quantification in HiLog (Chen et al. 1993) and F-logic (Kifer et al. 1995), and due to the proliferation of Skolemized head-existentials that are passed down to the engine by the knowledge acquisition system.

The first source of non-termination is the use of recursion, which plagues Prolog under the standard evaluation strategy. This can be illustrated by the following simple rule:

\[ p(X) \leftarrow p(X). \]
\[ \text{?- p(a).} \]

The prevalent way to address this problem is to use tabling, which is also known under the more technical term of SLG-resolution. Tabling was pioneered by the XSB system (Swift and Warren 2012) and is now supported by a number of other systems, such as Yap (Costa et al. 2012), B-Prolog (Zhou 2012), and Ciao (Hermenegildo et al. 2012). In the above example, tabling the predicate \( p \) will cause the evaluation to terminate.

The second reason for non-termination, even under SLG, is the pattern of increasingly deep nested calls generated during the evaluation, as in the following example:

\[ p(X) \leftarrow p(f(X)). \]
\[ \text{?- p(a).} \]

Since neither call subsumes the other, tabling will not help terminate the evaluation process. However, a surprisingly simple technique known as subgoal abstraction, also pioneered by XSB, takes care of this problem. The idea is to modify the calls by “abstracting” deeply nested subterms and replacing them with new variables. For instance, in the above example, we could abstract calls once the depth limit of 4 has been reached. As a result, \( p(f(f(f(f(a)))))) \) and all the subsequent calls would be abstracted to \( p(f(f(f(f(X))))). \) Tabling enhanced with subgoal abstraction is able to completely evaluate all queries that have finite number of answers.

The remaining major source of non-termination is when the number of answers to the query or its subqueries is infinite:

\[ p(a). \quad p(f(X)) \leftarrow p(X). \]
\[ \text{?- p(X).} \]

This query has the infinite number of answers: \( p(a), p(f(a)), p(f(f(a))), \ldots \)

Clearly, such queries cannot be evaluated completely, but if the program is what the user intended, the user could ask the system to stop after getting the required number of answers.

\(^3\) flora.sourceforge.net
answers. However, in our experience, the user usually does not intend to construct infinite predicates. Finding out how the infinite number of answers came about and fixing the problem is difficult even for an experienced programmer and even for programs that have just a few dozens of rules. For knowledge bases that have thousands of rules, like the ones we have been dealing with in the SILK project, diagnosing this problem is an onerous and frustrating job. In the absence of subgoal abstraction, this difficulty also exists for the aforesaid problem of detecting sequences of subgoals of increasing depth.

We remind that neither the problem of program termination nor that of whether the number of answers is finite is decidable (Schreye and Decorte 1994; Sipser 1996), so no algorithm can prove termination or non-termination in general. Sufficient conditions for termination of logic programs have been proposed in the literature (Schreye and Decorte 1994; Verbaeten et al. 2001; Lindenstrauss et al. 2004; Bruynooghe et al. 2007; Nguyen and De Schreye 2007; Nguyen et al. 2008; Schneider-kamp et al. 2010), but most deal with Prolog or Prolog-like evaluation strategies. Although many of these results are very deep, their practical impact is limited due to the punishing syntactic restrictions for which these termination tests work. Moreover, neither tabling nor subgoal abstraction are taken into account, so these works have limited use for advanced logic engines like XSB and its derivatives, Flora-2 and SILK. In a recent work, (Liang and Kifer 2013) took a different approach and developed techniques to enable users to analyze the causes of non-termination. We proposed a suite of algorithms, called the non-termination analyzer or Terminyzer, which was able to detect sequences of tabled subgoal calls and functor applications that are the potential causes of non-termination. These algorithms analyze forest logging traces and output sequences of tabled subgoal calls that form non-terminating call-cycles. Unfortunately, in many cases this feedback was hard to use in practice, as it was fairly imprecise. The same subgoal could occur in multiple rule heads and in even more places in rule bodies, so Terminyzer left too much tedious work for the user to do manually.

Here we propose a new suite of algorithms, Terminyzer+, which closes the usability gap or, at least, narrows it down significantly. Terminyzer+ can detect not only sequences of subgoals that cause non-termination, but, importantly, the exact rules where these calls occur and the rule sequences that are fired in a cyclic manner, which lead to non-termination. This makes Terminyzer+ amenable to serving as a back-end for user-friendly graphical interfaces, which can greatly simplify the debugging process. Such an interface is currently under construction in the SILK project.

The key idea that makes Terminyzer+ possible is a program transformation that assigns a unique rule id to each rule and modifies the rules in such a way that the id information is preserved in the log forest trace. The transformation increases the size of each subgoal slightly, by adding an additional argument. Furthermore, our new algorithms do not depend on subgoal abstraction, although they can take subgoal abstraction into account, if the underlying engine supports it. Finally, we make a step towards automatic remediation of non-termination programs by proposing an algorithm that heuristically fixes some non-terminating programs.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 presents the transformation that adds a unique id to each rule. Section 4 describes Terminyzer+ for tabled logic engines with subgoal abstraction. Section 5 presents auto-repair techniques for certain non-terminating behaviors. Section 6 extends
Terminyzer+ to tabled logic engines that do not support subgoal abstraction. Section 7 presents experimental studies, and Section 8 discusses related work and concludes the paper.

2 Preliminaries

2.1 Tabling in XSB

The limitations of Prolog’s standard SLD resolution-based evaluation strategy are well-known: it is incomplete and can go into an infinite loop even for simple Datalog rule sets. To address this problem, SLG resolution (also known as “tabling”) was developed over 20 years ago and (Swift and Warren 2012) provides the most recent insight into this mechanism.

In tabled evaluation, calls to tabled predicates are cached in a table $T$ for subsequent calls. $T$ can be viewed as a set of pairs of the form $(sub, ansrs)$ where ansrs are proven instances of sub. When a tabled subgoal, sub, is issued, SLG examines whether there is a pair $(sub', ansrs') \in T$ such that sub is similar (to be explained shortly) to sub'. If so, then answer clause resolution is performed instead of program clause resolution, i.e., ansrs' are used to satisfy sub. In this case, sub is referred to as the consumer of sub' while sub' is the producer of sub. Otherwise, a new table entry of the form $(sub, ansrs)$ is added to $T$, where initially ansrs = ∅. Then sub is resolved against program clauses as usual in Prolog. All newly derived answers for sub are added to ansrs, sub becomes a producer of these answers, and all subsequent subgoals that are similar to sub become consumers of sub's answers.

There are two main ways to define subgoal-similarity mentioned above. Depending on which notion is chosen, the tabling strategy is called variant or subsumptive. In variant tabling, sub is similar to sub' if sub is a variant of sub', i.e., they are identical up to variable renaming. In subsumptive tabling, sub is similar to sub' if sub is subsumed by sub', i.e., there is a variable substitution $\sigma$ such that $\sigma(sub') = sub$. Note that in this case the notion of similarity is asymmetric. Since only unique answers are added to the table and returned to consumers, tabled evaluation terminates if there is only a finite number of tabled subgoals and each tabled subgoal has finitely many answers. For instance, this is the case in Datalog, i.e., when function symbols are not present. It has been proven that tabled evaluation terminates for any program with the bounded term depth property, i.e., all terms that are ever generated in the course of SLG resolution, including all subgoals and answers, have an upper bound on their depth (Swift and Warren 2012).

2.2 Forest Logging in XSB

The workings of SLG resolution can be captured by an SLG forest, which has an SLG tree for every new (dissimilar) subgoal to a tabled predicate. The SLG tree for sub has root of the form $sub :- sub$, and each non-root node is of the form $\theta(sub) :- \theta(leftsubs)$, where $\theta$ is the substitution obtained from resolving sub against the knowledge base and $\theta(leftsubs)$ are the remaining subgoals needed to prove sub. If $\theta(leftsubs)$ is an empty clause, $\theta(sub)$ is an answer to sub. Children of a root node are obtained through resolution of a tabled subgoal against program clauses. Children of non-root nodes are obtained...
Terminyzer+: An Automatic Non-Termination Analyzer for Logic Programs

through answer clause resolution if the left most selected literal is tabled (e.g. children of node 11 in the tree for \texttt{path(1,Y)} of Example 1), or through program clause resolution if the leftmost selected literal is not tabled (e.g. children of nodes 2 and 5 in the tree for \texttt{path(1,Y)} of Example 1). Each edge in the tree corresponds to a derivation step of program or answer clause resolution.

\textit{Example 1}

The SLG forest for the following XSB program is shown in Figure 1, where each node is labeled with an ordinal denoting the creation order of the node during evaluation.

\begin{verbatim}
:- table path/2.
edge(1,2). edge(1,3). edge(2,1).
path(X,Y) :- edge(X,Y). path(X,Y) :- edge(X,Z), path(Z,Y).
?- path(1,Y).
\end{verbatim}

Fig. 1. The SLG Forest for Example 1

After the subgoal \texttt{path(1,Y)} in node 11 uses all the answers from its producer in node 1, the evaluation of \texttt{path(2,Y)} in node 11 is suspended since its producer in node 1 is not yet completely evaluated. The evaluation of \texttt{path(1,Y)} in node 6 is suspended due to node 7 for similar reasons, and an alternative derivation is tried at node 14. At node 17, all possible derivation alternatives have been exhausted, and the suspended evaluations are resumed and completed. This is a simplified version of an example in (Swift et al. 2013). □

Compared to Prolog systems, logic engines that support tabling are much more involved. They suspend and resume computation paths, delay negated subgoals that are involved in loops through negation, simplify these subgoals once their truth values become known, and manage the table accordingly. For debugging and performance optimization, programmers may need to inspect table operations during evaluation. To this end, XSB has recently provided a new facility, called \textit{forest logging} (\texttt{logforest}), which makes the table events available to the programmer. These events include:
Call to a tabled predicate. If a subgoal parent calls another subgoal child, i.e., the evaluation of parent fires a rule that issues child, a Prolog fact of the form \(tc(\text{child}, \text{parent}, \text{status}, \text{timestamp})\) is logged. Here \(\text{timestamp}\) is the timestamp of the event representing its sequence order and \(\text{status}\) is the current status of child. It can take the following values:

- \(\text{new}\) if child is a newly issued subgoal;
- \(\text{cmp}\) if the evaluation of child has been completed; and
- \(\text{incmp}\) if child is not a new subgoal, but is yet to be completely evaluated.

If the subgoal is negative, a similar fact \(\text{nc}(\text{child}, \text{parent}, \text{status}, \text{timestamp})\) is logged. If child is the first tabled subgoal in an evaluation, parent is root.

Derivation of a new answer. When a new answer, \(\text{ansr}\), is derived for sub and added to the table, the fact \(\text{na}(\text{ansr}, \text{sub}, \text{timestamp})\) is added to the log. When a new conditional answer \(\text{ans} :\text{-} \text{delayed literals}\) is derived for sub and added to the table, a log of the form \(\text{nda}(\text{ansr}, \text{sub}, \text{delayed literals}, \text{timestamp})\) is recorded. Here \(\text{ans}\) is the answer substitution and \(\text{delayed literals}\) are the delayed literals whose truth value is yet to be determined (this usually occurs due to recursive loops through negation).

Return of an answer to a consumer. If an answer \(\text{ansr}\) is returned to a consumer child which is called by parent, the fact \(\text{ar}(\text{ansr}, \text{child}, \text{parent}, \text{timestamp})\) is added to the log. If the sequence is conditional, \(\text{dar}(\text{ansr}, \text{child, parent, timestamp})\) is recorded.

Subgoal completion. When a set of mutually recursive subgoals, \(S\), are completed, \logforest\ records \(\text{cmp}(\text{sub}, \text{sccnum}, \text{timestamp})\) for each \(\text{sub} \in S\), where \(\text{sccnum}\) is an ordinal that identifies \(S\). If \(\text{sub}\) is completed early (i.e., its truth is established without the need to fully evaluate all the dependent subgoals), \(\text{cmp}(\text{sub}, \text{ec, timestamp})\) is logged where \(\text{ec}\) stands for early completion.

Other events. \logforest\ also records delays of negative literals, table abolishes, and errors. These events are not needed for our purposes and we will omit them in the sequel.

Example 2
For the SLG forest of Example 1, the \logforest\ trace is given in the first column of Table 1. The second column in the table is the label of the node in the trees of Figure 1 where a corresponding event happens. The third column is an explanation. An answer for a subgoal is represented as a substitution for the list of variables in the subgoal. For instance, in the second log entry \(\text{na}([2], \text{path}(1, x0), 1)\), the answer is represented as \([2]\) and the list of variables in the subgoal \(\text{path}(1, x0)\) are \([x0]\). It means that the substitution \(x0 = 2\) is an answer.

3 Adding Ids to Rules
The key enabling idea in \Terminyzer+\ is a transformation that adds unique ids to rules in such a way that this information is preserved in the forest logging trace. For our purposes, we want each subgoal call in the trace to “remember” the rule from which this call was issued. Although this information is not available in the original program, it is
easy to instrument any logic program so that each subgoal call would be stamped with the id of its host rule (i.e., rule from whose body the call was issued).

The transformation described in Algorithm 1 processes the original program rule by rule and assigns a new id to each newly encountered rule. It creates a new rule $\bar{R}$ out of a rule $R$, where each tabled predicate, $p/N$, is augmented with one more argument, so that $p/N$ is replaced with $p/(N+1)$.

Queries are changed as follows: if the query predicate is not tabled, the query is not changed. If that predicate is tabled, one more (last) argument is added, which contains a new variable.

It is easy to see that the new program is equivalent to the original one in the sense that untabled queries to both programs have the same answers and the answers to the tabled predicates are the same if the last component in each answer tuple is chopped.
while unprocessed rules remain do
get the next program rule $R$: \( h(t_1,\ldots,t_k) :- \text{body}; \)
generate a new rule id, $id(R)$;
if $h/k$ is tabled then change the head literal to $h(t_1,\ldots,t_k,\text{Newvar})$;
else leave the head literal unchanged;
replace each tabled subgoal, $p(s_1,\ldots,s_m)$, in body with $p(s_1,\ldots,s_m, id(R))$;
end

Algorithm 1: Program Transformation: Adding Rule Ids

However, now each subgoal call recorded in the log will be labeled with the id of the rule from which this call was issued. For instance, the following rule and a query

\[
\text{r}(X) :- p(X), s(X,Y), q(Y).
\]

\(?- \text{r}(a).\)

get transformed into the following, assuming that the assigned rule id is 123, that $r/1$, $p/1$, and $q/1$ are tabled, and that $s/2$ is not:

\[
\text{r}(X,\_):- p(X,123), s(X,Y), q(Y,123).
\]

\(?- \text{r}(a,\_).\)

\text{FLORA-2} and SILK currently support rule ids, although the form of the last argument there is more complex, which is used to provide additional support for truth maintenance.

4 Termynzer+ for Tabled Logic Engines with Subgoal Abstraction

This section extends the call sequence analysis and answer flow analysis approaches in (Liang and Kifer 2013) for tabled logic engines that support subgoal abstraction. The analysis assumes that execution is stopped after a time limit set by the user or after the evaluation starts producing answers that exceed certain size limits (e.g., term depth), and then analyzing the logs. Our examples assume that the system stops when it generates query answers of depth greater than 10.

We should stress that due to the undecidability results mentioned in Section 1, one cannot algorithmically prove non-termination in all cases unless infinite logs are available. Pragmatically, this means that, in working with Termynzer+, one must assume that the available logs are “long enough.”

4.1 Call Sequence Analysis

Recall that with subgoal abstraction, the only way for tabled query evaluation to not terminate is when the query or its subgoals have infinitely many answers. Call sequence analysis, in this case, finds the exact sequence of subgoal calls to tabled predicates and, for each subgoal, its host rule’s id. Moreover, it identifies the potential sets of recursive predicates that are causing non-termination.

As presented in Section 2, when a tabled subgoal \textit{sub} has been completely evaluated and all its answers have been recorded in the table, \texttt{logforest} records a log entry of the

\footnote{We assume that the programs have no aggregate functions such as \texttt{count}, which are sensitive to duplicate answers.}
form \( \text{cmp}(\text{sub}, \text{sccnum}, \text{timestamp}) \). We say that such calls are \textit{finished}. Otherwise, the call is \textit{unfinished} and can be found via the following rule:

\[
\text{unfinished}(\text{Child}, \text{Parent}, \text{Timestamp}) :-
\text{tc}(\text{Child}, \text{Parent}, \text{Stage}, \text{Timestamp}); \text{nc}(\text{Child}, \text{Parent}, \text{Stage}, \text{Timestamp}),
\text{Stage == new ; Stage == incmp},
\text{not_exists}(\text{cmp}(\text{Child}, \text{SCCNum}, \text{Timestamp}1)).
\]

Here \textit{not_exists} is the XSB well-founded negation operator, which, in this case, existentially quantifies \text{SCCNum}, \text{Timestamp}1.

The fact \textit{unfinished}(\text{child}, \text{parent}, \text{timestamp}) says that unfinished subgoal \text{child} is called by \text{parent} and the event timestamp is \text{timestamp}. Since \text{parent} is waiting for the answers from \text{child}, \text{parent} is a child of another unfinished subgoal. The initial subgoal, \text{root}, has no parent. The sequence of all unfinished calls, sorted by their \textit{timestamps}, is the exact sequence of calls that caused non-termination. The rules that issued each of these unfinished subgoals are also easily identified: they are recorded as the last argument of a subgoal, as is seen from the transformation in Section 3.

**Theorem 1 (Soundness of Call Sequence Analysis)**

Consider a query to a program all of whose predicates are tabled, and assume that the system supports subgoal abstraction. If there exist unfinished calls in its complete infinite forest logging trace, then the computation is non-terminating.

Theorem 1 obviously holds, because all calls should have been completed if the evaluation terminates. However, as mentioned earlier, one cannot obtain the complete infinite trace for a non-terminating evaluation. In practice, one would let the program execute long enough until it starts producing answers exceeding some size limits and then analyze the available portion of the log. We will now turn to the machinery for detecting non-terminating behavior by analyzing forest logging traces.

The \textit{unfinished-call child-parent graph} (CPG) for a forest logging trace is a directed graph \( G_{uc} = (\mathcal{N}, \mathcal{E}) \) whose nodes are \( \mathcal{N} = \{ \text{child} \mid \text{unfinished}(\text{child}, \text{parent}, \text{timestamp}) \} \cup \{ \text{root} \} \). Subgoals that are variants of each other (i.e., identical up to the variable renaming) are treated as the same subgoal. Each \text{sub} \in \mathcal{N} is labeled with the timestamp of the first call to \text{sub}; it is written as \text{sub.timestamp}. The timestamp of initial subgoal \text{root}, \text{root.timestamp}, is -1. A directed edge \((\text{sub}_1, \text{sub}_2)\) is in \( \mathcal{E} \) if and only if \text{sub}_1 is an unfinished parent-subgoal of \text{sub}_2, i.e., \text{unfinished}(\text{sub}_2, \text{sub}_1, \text{timestamp}) is true. The edge that corresponds to \text{unfinished}(\text{sub}_2, \text{sub}_1, \text{timestamp}) is labeled with the timestamp of this fact, denoted \((\text{sub}_1, \text{sub}_2).\text{timestamp}\). The timestamps of nodes and edges preserve the temporal order of their creation in forest logging trace.

An \textit{unfinished-call path} is a path with no repeated edges in \( G_{uc} \); it is called an \textit{unfinished-call loop} if it is a cycle. An unfinished-call path of the form \([\text{sub}, \text{sub}]\) means that there is an edge \((\text{sub}, \text{sub})\) in \( \mathcal{E} \) and it is also an unfinished-call loop. Loops that represent the same cycles in CPG are considered to be the same and we keep only one representative for each set of such loops. For instance, \([a, b, c, a]\) and \([b, c, a, b]\) are the same loop while \([a, b, c, a]\) and \([a, c, b, a]\) are not. Unfinished-call loops contain recursive subgoals that are potential causes of non-termination.

**Example 3**

Consider the following non-terminating query where \@!sym indicates the id of the corresponding rule:
produces logs containing these unfinished calls:

\[
\begin{align*}
\text{unfinished}(r(_h60,_h68), \text{root}, 0) \\
\text{unfinished}(r(_h30,r1), r(_h30,_h49), 8) \\
\text{unfinished}(r(_h00,r1), r(_h00,r1), 11) \\
\text{unfinished}(p(_h70,r4), r(_h70,r1), 12) \\
\text{unfinished}(q(_h40,r2), p(_h40,r4), 16) \\
\text{unfinished}(p(_h10,r3), q(_h10,r2), 20) \\
\text{unfinished}(q(_h80,r2), p(_h80,r3), 24)
\end{align*}
\]

This is the exact sequence of calls causing non-termination. There are 6 unfinished subgoals as shown in Figure 2(A), where each subgoal’s timestamp and the host rule’s id are also given. The unfinished-call CPG has 7 edges, shown in Figure 2(B), where timestamps are used to represent nodes instead of actual subgoals and each edge is labeled with its timestamp. There are 2 unfinished-call loops: \([8, 8]\) and \([16, 20, 16]\). 

Algorithm 2 constructs the unfinished-call CPG \(G_{uc} = (N, E)\) from the set of unfinished calls of a forest logging trace. For each \(\text{unfinished}(\text{child}, \text{parent}, \text{timestamp})\), the node \(\text{child}\) is added, if it does not already exist, plus the edge \((\text{parent}, \text{child})\). All unfinished calls are processed in the order of their creation in forest logging, which is also the order in which these unfinished calls are made during evaluation. Thus, when \(\text{unfinished}(\text{child}, \text{parent}, \text{timestamp})\) is encountered we know that \(\text{parent}\) must have been added to the graph as an child-subgoal of its parent, i.e., \(\text{unfinished}(\text{parent}, p', c')\) must be true for some \(p'\) and \(c' < \text{timestamp}\). We have two cases:

1. \(\text{child} \in N\). The evaluation calls a previously issued subgoal.
2. \(\text{child} \notin N\). A new subgoal is called and a new node is added to the graph.

In the first case, an unfinished-call loop exists, so the current evaluation path of \(\text{parent}\) is suspended and alternative derivations will be explored. This implies an important property of unfinished-call CPGs: an unfinished-call loop is created out of an (acyclic)
path always by adding a final edge of the form \((\text{sub}_1, \text{sub}_2)\), where \(\text{sub}_1.\text{timestamp} \geq \text{sub}_2.\text{timestamp}\). We call such an edge a critical loop edge, such as the edges labeled with 11 and 24 in Figure 2 of Example 3.

1. Let \(UC\) be the set of all unfinished calls;
2. \(E = \emptyset; \mathcal{N} = \{\text{root}\}; \text{root.timestamp} = -1;\)
3. \textbf{while} \(UC \neq \emptyset\) \textbf{do}
4. \textbf{Remove} unfinished \((\text{child}, \text{parent}, \text{timestamp})\) from \(UC\), where \(\text{timestamp}\) is the smallest among \(UC\);
5. \textbf{if} \(\text{child} \notin \mathcal{N}\) \textbf{then}
6. \(\mathcal{N} = \mathcal{N} \cup \{\text{child}\}; \text{child.timestamp} = \text{timestamp};\)
7. \(E = E \cup \{(\text{parent}, \text{child})\}; (\text{parent}, \text{child}).\text{timestamp} = \text{timestamp};\)
8. \textbf{return} \(G_{uc} = (\mathcal{N}, E)\)

Algorithm 2: Unfinished-Call CPG Construction

If critical loop edges are taken out, any unfinished-call CPG becomes a connected directed acyclic graph (i.e., a tree) in which every edge goes from a node with a smaller timestamp to a node with a larger timestamp. A path, connecting the root node to a subgoal, can be computed as \(\text{uc.path(Sub,Path)}\) by the following rule set.

\begin{verbatim}
:- table reversed_uc_path/2.
uc_path(C,P) :- reversed_uc_path(C,RevP), reverse(RevP,P).
reversed_uc_path(C,[C,root]) :- unfinished(C,root,\_).
reversed_uc_path(C,[C|P]) :- unfinished(C,Parent,\_),
       Parent.timestamp<C.timestamp, reversed_uc_path(Parent,P).
\end{verbatim}

After computing all unfinished-call paths, without critical loop edges, from \(\text{root}\) to other nodes, all distinct unfinished-call loops can be computed by checking whether there exists a critical loop edge from the last vertex of a path to any other node in the same path. Consider an unfinished-call path \(P = [\text{root}, \text{sub}_1, \ldots, \text{sub}_n]\). If there is a critical loop edge \((\text{sub}_i, \text{sub}_j), 1 \leq i \leq n\), then the part of \(P\) from \(\text{sub}_i\) to \(\text{sub}_n\), \([\text{sub}_i, \ldots, \text{sub}_n]\), is an unfinished-call loop.

**Theorem 2** (Completeness of Call Sequence Analysis)
Consider a query and a program all of whose predicates are tabled and assume that the system supports subgoal abstraction. If the evaluation does not terminate, then there is at least one unfinished-call loop in the unfinished-call child-parent graph constructed for complete infinite forest logging trace. □

**Proof of Theorem 2**
Suppose there is no unfinished-call loop in the corresponding unfinished-call CPG \(G_{uc} = (\mathcal{N}, E)\). There can be only a finite number of calls to tabled predicates because of subgoal abstraction, so \(G_{uc}\) would then be a finite graph. Since there is no unfinished-call loop, there must be terminal nodes that have no outgoing edges. Let \(S \subseteq \mathcal{N}\) denote this set of nodes. It means that their SLG-children are not in \(\mathcal{N}\), i.e., they are not unfinished subgoals. Therefore the SLG-children of \(S\) are either completely evaluated tabled subgoals or base facts. But then, after long enough time, all subgoals in \(S\) should have been completely evaluated and completed. This contradicts the assumption that \(S \subseteq \mathcal{N}\), i.e., the subgoals in \(S\) are unfinished. □
As mentioned, since complete infinite traces for non-terminating computations cannot be had, in practice one would let the program execute long enough until it starts producing answers exceeding some size limits, and then analyze the available portion of the log.

4.2 Answer Flow Analysis

Call sequence analysis finds the exact sequence of subgoal calls and the corresponding host rules that are responsible for non-termination. It also identifies the potential sets of recursive predicates that form the unfinished-call loops and cause generation of infinitely many answers. However, there can be exponentially many unfinished-call loops in an unfinished-call CPG, and not all such loops may be causing non-termination. For instance, there are two unfinished-call loops in Figure 2, but only \([16, 20, 16]\) is at fault. We say that an unfinished-call loop is a culprit if it is a cause for non-termination.

Answer flow analysis looks for the log entries that specify the answers being returned to parents (the \(ar\)-facts and \(dar\)-facts) at the end of the logforest trace and produces child-parent relationships among unfinished subgoals. These child-parent relationships help us identify precisely which unfinished-call loops are culprits, so we could track how answers percolate through the unfinished subgoals.

If there are infinitely many answers, each new answer, \(ansr\), to an unfinished subgoal, \(sub\), is returned to the parents of \(sub\) and these parents use \(ansr\) to derive their own answers. The newly derived answers for the parents of \(sub\) are returned to the parents of the parents, and this gives rise to an endless process in which subgoals continue to receive, derive and return answers. An answer-flow child-parent sequence is the sequence of child-parent pairs found in all the log entries for answers returned to parents; it captures the child-parent relationships in the above endless process. The pairs of an answer-flow child-parent sequence are sorted by their creation order (timestamp). A child might continue returning multiple answers to a certain parent before the parent starts deriving its own answers. In this case, only one child-parent pair is recorded for all such answer returns, since all these pairs are identical.

Since there can be only a finite number of unfinished subgoals because of subgoal abstraction, the answer-flow child-parent sequence of a non-terminating trace must contain repetitions (Theorem 3). An answer-flow child-parent sequence, \(cps\), if \(cpp\) is a subsequence of \(cps\) that repeats at least twice at the end of \(cps\), i.e., \(cps = \ldots \bullet cpp \bullet cpp\), where \(\bullet\) is the sequence concatenation operator. For instance, \([\langle c_2, p_2 \rangle, \langle c_3, p_3 \rangle]\) is a child-parent pattern of length two in \([\langle c_1, p_1 \rangle, \langle c_2, p_2 \rangle, \langle c_3, p_3 \rangle, \langle c_2, p_2 \rangle, \langle c_3, p_3 \rangle]\). Thus, the child-parent sequence \(cps\) of a non-terminating trace has a prefix that does not end with a \(cpp\) and a suffix that consists of two or more repeated \(cpp\)'s, i.e., \(cps = prefix \bullet cpp^n\) where \(n > 1\) and \(cpp^n\) represents the concatenation of \(n\) \(cpp\)'s. We will call this suffix \(cpp^n\) the \(cpp\)-suffix of that \(cps\). The optimal child-parent pattern in a child-parent sequence \(cps\) is the shortest child-parent pattern, \(cpp\), such that its \(cpp\)-suffix is the longest in \(cps\) (among all suffixes of child-parent patterns in \(cps\)). We use \(optimal_{cpp}(child, parent)\) to denote the fact that \((child, parent)\) is in the optimal child-parent pattern. We will see that non-termination implies the existence of an optimal child-parent pattern (Theorem 3).

Given a child-parent sequence, let \(pat\) be the subsequence containing the last \(n\) elements...
in the sequence. The predicate \texttt{pattern(cps, len, pat, times)} specifies the number of times a child-parent pattern \textit{pat} of length \textit{len} repeats at the end of \textit{cps}. Patterns of different lengths can be computed by posing the query \texttt{?- pattern(cps, len, Pat, Times)} to the following rules, where the \textit{len} parameter successively assumes the values 1, 2, and so on. In this way, we will either find an optimal child-parent pattern or determine that there is no pattern.

\begin{verbatim}
pattern(CPS,Len,Pat,Times) :-
    length(Pat,Len),
    %% This binds Pat to the suffix of CPS of length Len
    append(CPSPrefix,Pat,CPS),
    aux_pattern(CPSPrefix,Pat,Times).

aux_pattern(CPS,Pat,Times) :-
    append(CPSPrefix,Pat,CPS), !,
    pattern(CPSPrefix,Pat,TimesPrefix),
    Times is TimesPrefix+1.

aux_pattern(CPS,Pat,1).
\end{verbatim}

\textbf{Example 4}

The child-parent sequence of the forest logging trace for Example 3 is the \textit{cps} below:

\begin{verbatim}
[(q(_h599,r2),p(_h599,r3)), (p(_h599,r3),q(_h599,r2)), (q(_h619,r2),p(_h619,r3)),
 (p(_h639,r3),q(_h639,r2)), (q(_h659,r2),p(_h659,r3)), (p(_h679,r3),q(_h679,r2)),
 (q(_h699,r2),p(_h699,r3)), (p(_h719,r3),q(_h719,r2)), (q(_h739,r2),p(_h739,r3)),
 (p(_h759,r3),q(_h759,r2)), (q(_h779,r2),p(_h779,r3))].
\end{verbatim}

This \textit{cps} has two child-parent patterns. The first is \textit{cpp\textsubscript{1}} = \[(p(_h759,r3),q(_h759,r2)),
(q(_h779,r2),p(_h779,r3))\] of length two; it repeats five times. The second is \textit{cpp\textsubscript{2}} = \textit{cpp\textsubscript{1}} of length four, which repeats twice. The optimal child-parent pattern is \textit{cpp\textsubscript{1}}, as it covers 2 \times 5 = 10 entries in \textit{cps} compared to \textit{cpp\textsubscript{2}}, which covers only 4 \times 2 = 8 entries. □

Let \textit{cpp\textsubscript{opt}} be the optimal child-parent pattern for the forest logging trace in question. As in the call sequence analysis, child-parent relationships in \textit{cpp\textsubscript{opt}} are modeled as a graph. An answer-flow CPG for a forest logging trace is a directed graph \(G_{af} = (N, E)\), defined as follows. \(N\) is the set of children and parent-subgoals in \textit{cpp\textsubscript{opt}}, i.e., \(N = \{\text{sub} | (\text{sub},...) \in \text{cpp\textsubscript{opt}} \text{ or } (...\text{sub}) \in \text{cpp\textsubscript{opt}}\}\). Edges in \(G_{af}\) are the child-parent pairs in \textit{cpp\textsubscript{opt}}, i.e., \(E = \{(\text{child},\text{parent}) | (\text{child},\text{parent}) \in \text{cpp\textsubscript{opt}}\}\).

A path in \(G_{af}\) is called an answer-flow path; such a path is called an answer-flow loop if it is a cycle. Two answer-flow loops that consist of the same nodes and edges are considered to be the same and we will keep only one representative loop in such a case. Answer-flow paths and loops represent information flow among unfinished subgoals in the infinite process of answer derivation. They can be computed in a way similar to the computation of unfinished-call paths and loops. All answer-flow paths from node \textit{child} to node \textit{parent} can be computed using the predicate \texttt{af\_path(child, parent, path)}; all answer-flow loops starting from \textit{child} can be computed using the predicate \texttt{af\_loop(child, loop)}, defined below.

\begin{verbatim}
:- table af\_path/3.
af\_path(Child,Parent,[Child]) :- optimal\_cpp(Child,Parent).
af\_path(Child,Parent,[Child|P]) :- optimal\_cpp(Child,Sub),
    af\_path(Sub,Parent,P), \+ member(Child,F).
af\_loop(Sub,Loop) :- af\_path(Sub,Sub,Loop).
\end{verbatim}
Example 5
Consider $cpp_1$, the optimal child-parent pattern of Example 4. Its answer-flow graph is the subgraph shown inside the rectangle in Figure 2. The only answer-flow loop is $[16, 20, 16]$, which tells us that subgoal $p$ called from rule $r3$ and subgoal $q$ called from rule $r2$ return answers to each other in an infinite answer derivation loop. □

Theorem 3 (Completeness of Answer Flow Analysis)
Consider a query to a program all of whose predicates are tabled and assume that the inference engine supports subgoal abstraction. If the query evaluation does not terminate, then:

i. There is an optimal child-parent pattern in its complete infinite trace,

ii. $G_{af} = (N, E)$ contains at least one answer-flow loop, and

iii. Every $\text{sub} \in N$ appears in at least one answer-flow loop. □

Proof of Theorem 3
(i) There can be only a finite number of unfinished subgoals due to subgoal abstraction, and thus there must be at least one child-parent pattern, for otherwise the evaluation would terminate. Therefore, there is an optimal child-parent pattern in the forest logging trace.

(ii) Suppose there is no answer-flow loop in $G_{af}$. There must be a set $S \subseteq N$ of terminal nodes and since these nodes are terminal, the graph has no edges going out of $S$. The SLG-children of these terminal nodes are therefore not in $N$ and answers for these SLG-children are not being repeatedly derived. Recall that, due to subgoal abstraction, there is only a finite number of nodes in $G_{af}$ and if we let the engine run long enough then all possible edges in $G_{af}$ will be generated and further computation will not change that graph. Therefore, the nodes for which answers are no derived repeatedly cannot stay unfinished (in the sense of unfinished SLG subgoals) infinitely long. So, after a while, all SLG-children of $S$ must either become completely evaluated tabled subgoals or they must have been base facts since the beginning. But this implies that, given enough time, all subgoals in $S$ should have been completed, which contradicts the assumption that $S \subseteq N$. Therefore, there must be an answer-flow loop.

(iii) If $\text{sub} \in N$ and $\text{sub}$ is not contained in any answer-flow loop, then its evaluation should have been completed and it cannot be in any child-parent pattern, a contradiction.

□

Theorem 4 (Soundness of Answer Flow Analysis)
Consider a query to a program all of whose predicates are tabled. If the complete infinite trace of that query has an optimal child-parent pattern then the query evaluation does not terminate. □

Theorem 4 follows from the definitions, since the optimal child-parent pattern captures the information flow among unfinished subgoals in a non-terminating computation. These theorems tell us that the set of subgoals contained in the optimal child-parent pattern of a non-terminating trace, i.e., the nodes of the pattern’s answer-flow CPG, are exactly the subgoals for which infinitely many answers continue being derived. We call these subgoals the culprit unfinished subgoals.
In call sequence analysis, an unfinished-call CPG is constructed and the suspected unfinished-call loops are flagged. Similarly, in answer-flow analysis, one builds answer-flow CPG and computes culprit loops, which shed light on how answers flow among culprit subgoals. The following Theorem 5 connects these two approaches.

**Theorem 5**

Let $G_{uc} = (N_{uc}, E_{uc})$ be the unfinished-call CPG and $G_{af} = (N_{af}, E_{af})$ be the answer-flow CPG for a non-terminating forest logging trace. Then $N_{af} \subseteq N_{uc}$, and for every edge $(child, parent) \in E_{af}$ there is an edge $(parent, child) \in E_{uc}$. Furthermore, any answer-flow loop is a culprit unfinished-call loop. □

**Proof of Theorem 5**

If $sub \in N_{af}$ then it must be an unfinished subgoal, since answers to $sub$ keep being derived, so the evaluation of $sub$ has not been completed. Thus, $N_{af} \subseteq N_{uc}$. In fact, we even have that $N_{af} \subset N_{uc}$, since $root \in N_{uc} \setminus N_{af}$. For any edge $(child, parent) \in E_{af}$, we know that $child$ returns answers to $parent$, i.e., it is issued in a SLG tree for $parent$. Therefore $(parent, child) \in E_{uc}$. This implies that any answer-flow loop is also an unfinished-call loop. □

## 5 Auto-Repair of Rules

In many cases, query evaluation does not terminate not because the query has infinitely many answers but because one of its subgoals does. In such cases, the query may terminate if a different evaluation order for its subgoals is used. This section describes one such heuristic technique to fix certain non-termination queries by delaying the evaluation of culprit subgoals.

Suppose that $G_{uc} = (N_{uc}, E_{uc})$ is the unfinished-call CPG of a non-terminating evaluation. For each $(parent, child) \in E_{uc}$, we know that the call to $child$ from $parent$ has not been completed. Moreover, we know:

- the host rule of this call, and
- the common set of the unbound arguments of $parent$ and $child$, which are also the arguments whose bindings are to be derived.

To reduce the possibility that $parent$ gets infinite number of bindings from $child$ and thus diminish the possibility of non-termination caused by that call to $child$, we can delay the evaluation of $child$ in the host rule until the aforesaid unbound arguments get bound. If later in the evaluation it is established that the arguments cannot be bound, the delay of $child$ ceases and the subgoal executed. Similar evaluation delays can be applied to all unfinished calls in $E_{uc}$.

**Example 6**

Consider the evaluation and unfinished-call CPG of Example 3, where edge $(12, 16)$ represents $(p(h70, r4), q(h40, r2))$. Their common set of arguments consists of their only argument, i.e., their first argument, and the rule id contained in $q(h40, r2)$ is $r2$. Therefore, our technique will delay the evaluation of $q(h40, r2)$ in the rule $r2$ until its first argument becomes bound. □
FLORA-2 and SILK support delay quantifiers of the form \texttt{wish}(\texttt{cond}) and \texttt{must}(\texttt{cond}), where \texttt{cond} is an and/or combination of \texttt{ground}(variables) and \texttt{nonvar}(variables). This is similar to the \texttt{when}/2 predicate found in many prologs with the difference being that the delayed subgoal is eventually tried even if the binding conditions are not met. A delayed literal is of the form \textit{delay-quantifier} \textit{goal}. When such a literal is to be executed, the attached \textit{delay-quantifier} is checked. If the quantifier’s condition is satisfied, \textit{goal} is executed immediately. Otherwise, the literal is delayed until such time that the condition is satisfied. If the condition is eventually satisfied, \textit{goal} is called. If the engine determines that satisfying the quantifier’s condition is impossible, \textit{goal} is called anyway (in case of the \texttt{wish} quantifier) or an error is issued (in case of the \texttt{must} quantifier).

Example 7
Consider the evaluation of Example 3. Our auto-repair technique will delay the unfinished subgoals and modify the program as follows:

\begin{verbatim}
@!f1 p(a). @!f2 q(b). @!f3 s(f1(b)).
@!r1 r(X) :- wish(ground(X))^r(X).
@!r2 p(f(X)) :- wish(ground(X))^q(X).
@!r3 q(g(X)) :- wish(ground(X))^p(X).
@!r4 r(X) :- wish(ground(X))^p(X), s(X).
?- wish(ground(X))^r(X).
\end{verbatim}

Such a program will then successfully terminate with an answer \(X = f1(b)\).

It should be clear, however, that the above technique is only a heuristic: no automatic fool-proof auto-repair technique is possible, in general.

6 Terminyzer+ for Tabled Logic Engines without Subgoal Abstraction
We now turn to non-termination analysis that does not rely on subgoal abstraction, which makes Terminyzer+ applicable to all tabling logic engines. As discussed in Section 1, in this case non-termination may be caused by creation of infinitely many subgoals. In this case, Terminyzer+ analyzes the sequence of unfinished subgoals and reports the predicates and their respective rule ids that form increasingly deep nested subgoals. As before, we assume that users stop the execution after a time limit or when subgoals or answers become too large.

For an unfinished subgoal, its simplified version is constructed out of the subgoal’s predicate and the rule id as \texttt{predicate(ruleid)}. For instance, \(p(f2(f1(a)),r3)\) is simplified to \(p(r3)\). The simplified unfinished subgoal sequence is the sequence of simplified unfinished subgoals sorted by the order of their first appearance in the trace. When non-termination happens due of an infinite number of subgoals, these subgoals must be increasingly deep. Since a finite program has only a finite number of predicates and functors, there must be repetitions in the aforesaid sequence of simplified unfinished subgoals.

Similarly to optimal child-parent pattern in Section 4.2, the optimal subgoal pattern of a simplified unfinished subgoal sequence can be computed. This pattern will show which subgoals in which rules call recursively one another and create increasingly deeper and deeper terms.
Example 8
The evaluation of this program

\[
\begin{align*}
@!f1 & \: p(a) . & \: @!f2 & \: s(a) . & \: @!r1 & \: r(X) :- r(X) . & \: @!r2 & \: p(X) :- q(f1(X)). \\
@!r3 & \: q(X) :- p(f2(X)) . & \: @!r4 & \: r(X) :- p(X), s(X).
\end{align*}
\]

?- r(a)

generates infinitely many unfinished subgoals, the first of which are

\[
\text{unfinished(r(a,,h46)), root, 0) } \quad \text{unfinished(r(a,r1), r(a,,h27), 8) } \quad \text{unfinished(r(a,r1), r(a,r1), 11)} \\
\text{unfinished(p(a,r4), r(a,,r1), 12)} \quad \text{unfinished(q(f1(a),r2), p(a,r4), 16)} \\
\text{unfinished(p(f2(f1(a)),r3), q(f1(a),r2), 19)} \quad \text{unfinished(q(f1(f2(f1(a))),r2), p(f2(f1(a)),r3), 22)} \\
\text{unfinished(p(f2(f1(f2(f1(a))),r3), q(f1(f2(f1(a)))),r2), 26)}
\]

whose simplified unfinished subgoal sequence has the prefix \{root, r(h46), r(r1), r(r1), p(r4), q(r2), p(r3), q(r2), p(r3)\}. The optimal subgoal pattern here is \{q(r2), p(r3)\}, which means that it is the predicates \(q\) in rule \(r2\) and \(p\) in rule \(r3\) that cause the generation of increasingly deep subgoals.

Theorem 6 (Soundness and Completeness)
Consider a query to a program all of whose predicates are tabled and assume that the system does not support subgoal abstraction. The forest logging trace has an optimal subgoal pattern if and only if the computation is non-terminating due to infinitely many subgoals.

Proof of Theorem 6
(Soundness) It is obvious that if there exists an optimal subgoal pattern then the evaluation does not terminate. Because if the evaluation terminates, there will be no unfinished subgoals and thus no optimal subgoal pattern. Suppose there are a finite number of subgoals. Since we know that there are only two causes of non-termination in tabled logic engines: infinitely many answers and infinitely many subgoals, the non-terminating behavior is thus caused by infinitely many answers. As described in Section 4.2, it means that the set of subgoals contained in the trace’s optimal child-parent pattern are receiving, deriving, and returning answers. It contradicts the fact that there exits an optimal subgoal pattern, which requires certain predicates from certain rules be recursively and repeatedly called.

(Completeness) As discussed above, when non-termination happens because of an infinite number of subgoals, these subgoals have to be increasing deep and formed by sets of recursive predicates together with function symbols, because otherwise there can only be a finite number of terms in a finite program. Therefore, there must be repetitions in the simplified unfinished subgoal sequence of its trace, which means there exists an optimal subgoal pattern.

Once the optimal subgoal pattern is computed, the user can easily debug his program since he would know which subgoals in which rules to look for. Note that without subgoal abstraction, the auto-repair technique presented in Section 5 does not apply.
7 Experiments

Terminyzer+ has been implemented for the FLORA-2 system, and we report our experiments below. All tests were performed on a dual core 2.4GHz Lenovo X200 with 3 gigabytes of main memory running Ubuntu 11.04 with Linux kernel 2.6.38.

Test programs. Here we include five test cases: T_1, T_2, T_3, T_4, and T_5, of which only T_1 terminates. The first four tests are performed with subgoal abstraction enabled, while T_5 was tested without subgoal abstraction. T_1 is the evaluation of Example 1. T_2 is the query and the rule set of Example 3. T_3 and T_4 are very large programs which were derived from FLORA-2 programs used in the SILK Project. T_3 has 844 rules and facts, and its corresponding XSB program (after FLORA-2-to-XSB translation) is estimated to have 2,000 rules and facts. T_4 consists of 4,774 rules and 919 facts, and its XSB program has over 1,000 facts and over 5,500 rules. T_5 is the program of Example 8.

For T_1, T_2, and T_3, we set XSB to abort after the answer depth reached the depth of 30. For T_4, we let the evaluation to continue until all available memory was consumed. The reason is that T_4 is a really complex program, and in order to get a usable prefix of its infinite trace, we have to let it run “long enough.” The Execution of T_3 produces a log trace of 3 megabytes with around 26,000 log entries, and the trace for T_4 is in excess of 2 gigabytes with more than 14 million log entries.

Test results. Terminyzer+ produced expected results in all the test cases. For T_1, it produced an empty unfinished-call CPG. For T_2, Terminyzer+ constructed the unfinished-call graph shown in Figure 2 and identified its culprit loop. The auto-repair technique presented in Section 5 successfully fixed the non-termination problem as demonstrated in Example 7.

For T_3, Terminyzer+ determined that the predicate entailed(X) of the following rule was generating infinitely many answers:

\[
\text{entailed}\left(\text{conjunction}(\text{Antecedent1}, \text{Antecedent2})\right) :- \\
\text{entailed}(\text{Antecedent1}), \text{entailed}(\text{Antecedent2}).
\]

The heuristic auto-repair method of Section 5 fails to fix this non-terminating query since it is the query itself, not its subqueries, that has infinitely many answers.

For T_4, the unfinished-call CPG has 14 nodes and 34 edges, and its answer-flow CPG has 9 nodes and 28 edges. Our auto-repair method successfully removes the cause of non-termination and the remedied program terminates with one answer. We should mention that an experienced knowledge engineer spent hours debugging T_4 — all in vein.

For T_5, Terminyzer+ successfully identified the optimal subgoal pattern, as described in Example 8.

Computation times. For T_1, T_2, T_3, and T_5, Terminyzer+ took less than 1 second for each program. For the much more complex T_4, it took 170 seconds. Compared to the hours spent by our knowledge engineer, Terminyzer+ appears to be a much more efficient alternative.

---

5 We also tested other, smaller, but still fairly large real programs from the SILK project with similarly positive results.
8 Related Work and Conclusion

There have been many studies on termination analysis for logic programs (Schreye and Decorte 1994; Verbaeten et al. 2001; Lindenstrauss et al. 2004; Bruynooghe et al. 2007; Nguyen and De Schreye 2007; Nguyen et al. 2008; Schneider-kamp et al. 2010) while non-termination analysis received much less attention (Decorte et al. 1998; Neumerkel and Mesnard 1999; Payet and Mesnard 2006; Voets and De Schreye 2009; Voets and De Schreye 2011). There are two major points that differentiate Terminyzer+. First, the termination and non-termination problems discussed in most previous work are non-issues in our framework, since they stem from incompleteness of the Prolog inference mechanism and, therefore, do not apply to our case. Second, Terminyzer+ utilizes traces to help the programmer debug his programs without syntactic restrictions. All other approaches perform static or dynamic analysis in order to prove termination and non-termination properties of restricted classes of logic programs.

Terminyzer+ extends our previous non-termination analyzer, Terminyzer (Liang and Kifer 2013), in several ways to make it practically useful for debugging large programs. First, Terminyzer+ gives more precise explanations by including rule ids from which the unfinished subgoals were called. This is extremely helpful even using textual reports produced by the system but, more importantly, the rule ids enable powerful graphical interfaces that can aid knowledge engineers. Second, we presented a heuristic technique for fixing non-terminating queries. Third, the use of rule ids made it possible to extend the analysis to systems that do not support subgoal abstraction. The usability of Terminyzer+ was confirmed by multiple experiments, which include two very large real-life programs.

Acknowledgments.

This work is part of the SILK (Semantic Inference on Large Knowledge) project sponsored by Vulcan, Inc. The authors were also partially supported by the NSF grant 0964196. We would like to thank Terry Swift for enabling this work by implementing subgoal abstraction and forest logging in XSB.

References


