GEM : a distributed goal evaluation algorithm for trust management

GEM: a Distributed Goal Evaluation Algorithm for Trust Management

Daniel Trivellato
TU/e Eindhoven
d.trivellato@tue.nl

Nicola Zannone
TU/e Eindhoven
n.zannone@tue.nl

Sandro Etalle
TU/e Eindhoven
University of Twente
s.etalle@tue.nl

ABSTRACT

Trust Management (TM) is an approach to distributed access control where access decisions are based on policy statements issued by multiple principals and stored in a distributed manner. Most of the existing goal evaluation algorithms for TM either rely on a centralized evaluation strategy, which consists of collecting all the relevant policy statements in a single location (and therefore they do not guarantee the confidentiality of intensional policies), or do not detect the termination of the computation (i.e., when all the answers of a goal are computed). In this paper we present GEM, a distributed goal evaluation algorithm for TM systems. GEM detects termination in a completely distributed way without the need of disclosing intensional policies, thereby preserving their confidentiality. We demonstrate that the algorithm terminates and is sound and complete w.r.t. the standard semantics for logic programs.

1. INTRODUCTION

Trust Management (TM) is an approach to access control in distributed systems where access decisions are based on the attributes of principals, attested by digital certificates (called credentials) [6]. Credentials are derived by means of policy statements issued by multiple principals and stored in a distributed manner; the set of policy statements issued by a principal forms the policy of that principal. In TM languages, policy statements are often represented as Horn clauses [17], possibly annotated with the *storage location* of the statement, which can be implicit [9, 18] or explicit [1, 4].

A distinguishing ingredient of TM is that all principals are free to define policy statements and to determine where to store them. Furthermore, the policy statements of a principal can refer to other principals' policies, thereby delegating authority to them. For instance, consider a scenario in which the Eindhoven hospital (EhvH) authorizes the members of project α run by the local pharmaceutical company C1 to access its medical laboratory. This corresponds to the policy statement *"(for each X) EhvH states that X is authorized to access the medical laboratory if X is a member of project α at C1"*. The statement can be expressed by the following clause:

$\mathsf{canAccessMedLab}(ehvH, X) \leftarrow \mathsf{memberOf}\alpha(c1, X).$

Here, EhvH relies on the policy statements of C1 to determine who is authorized to access the laboratory. To determine the authorized principals, it is clear that C1 needs to disclose to EhvH (part of) its *extensional policy*, i.e., the list of project members. However, in most existing TM systems C1 would also have to reveal to EhvH or to some other authority (part of) its *intensional policy*, that is, the policy statements used to determine the project members.

We argue that one of the advanced desiderata of TM systems is that the amount of information about policies that principals need to reveal to each other should be minimized. In fact, policies might contain confidential information about the relationships among principals that could be exploited by other principals in the domain (e.g., rival companies). Furthermore, the loss of policy confidentiality can result in attempts by other principals to influence the policy evaluation process [23]. Assume C1's policy to be the following:

$\mathsf{memberOf}\alpha(c1, X) \leftarrow \mathsf{projectPartner}(mc, Y), \mathsf{memberOf}\alpha(Y, X).$

This clause states that all the project members at partner companies in project α are also project members for C1. The list of partner companies is determined by the funding company MC, a multinational pharmaceutical company. If C1's policy was disclosed to other principals for evaluation (e.g., EhvH), the involvement of MC in project α along with the list of all project partners would become public. As a consequence, some competitors of MC could start investing on similar projects, limiting MC's competitive advantage, or could try to get a trusted individual hired by one of the partners companies to acquire sensitive information and results of the project. Preserving the confidentiality of intensional policies is therefore important not only to protect the relationships of the policy issuer, and more precisely the principals (and attributes) to which it delegates its authority, but also to limit the disclosure of the extensional policies of those principals.

Policy confidentiality is completely disregarded by some TM systems (e.g., [9, 18], where a central authority collects from the different agents all the clauses necessary to evaluate a given goal), and is only partially satisfied by others (e.g., [5, 16]). The reason why present systems do not satisfy this requirement is that in presence of mutually recursive policy statements it is very hard to detect when the computation has terminated without disclosing at least part of the intensional policy. To illustrate this point, let us assume the complete policy for the example above to be the following:

$\mathsf{projectPartner}(mc, c2).$
$\mathsf{projectPartner}(mc, c3).$
$\mathsf{projectPartner}(mc, c4).$
$\mathsf{memberOf}\alpha(c2, X) \leftarrow \mathsf{memberOf}\alpha(c1, X).$
$\mathsf{memberOf}\alpha(c2, alice).$
$\mathsf{memberOf}\alpha(c3, bob).$
$\mathsf{memberOf}\alpha(c4, charlie).$

To prevent the disclosure of intensional policies it is necessary to design a completely decentralized evaluation algorithm. Suppose that EhvH wants to prepare the access cards to the medical laboratory for the members of project α, and asks C1 for the list of project members. Instead of revealing its intensional policy to EhvH, C1 requests first the list of project partners to MC, and then the list of their project members to C2, C3 and C4; in turn, C2 poses the same request to C1. A first problem to be solved is that C1 should refrain from evaluating C2's request, as doing so could lead to a non-terminating chain of requests. This pitfall can be avoided using *tabling* [7].

However, the real challenge in designing a distributed algorithm in which intensional policies are not disclosed is to detect when all the answers have been collected. In our example, we have the following possible answer flow: C3 returns Bob as answer to C1, which forwards it to EhvH and C2; C4 returns Charlie as answer to C1; C1 sends Charlie as additional answer to EhvH and C2; C2 returns Alice, Bob, Charlie as answer to C1, which sends Alice as additional answer to EhvH and C2. At this point, all requests have been fully answered, but C1 does not know whether C2 will ever send additional answers. In other words, C1 is waiting for C2 to announce that its evaluation has terminated, and in turn C2 is waiting for C1 to announce that its evaluation has terminated. This situation is not acceptable in access control, where a decision (positive or negative) always needs to be taken. Detecting termination is also fundamental to allow for memory deallocation and the use of negation, which is employed by some TM systems to express non-monotonic constraints (e.g., separation of duty) [10, 12].

Another non-trivial issue in designing a distributed goal evaluation algorithm is determining *when* a principal should send the answers to a request. The simplest solution is to force each principal to send an answer as soon as it is computed. This is, however, suboptimal from the viewpoint of network overhead; in the example above, C1 eventually sends three distinct messages to EhvH and C2, one for each answer. A better solution would be for C1 to wait for the answers from C3 and C4 before sending its answers to the other principals. A naïve "wait" mechanism, on the other hand, might cause deadlocks; for instance, if C1 also waits for C2's answers, the computation deadlocks. It is therefore preferable to have a mechanism that allows principals to wait until they collected the maximum possible set of answers before sending them to the requester, while avoiding deadlocks.

In this paper we present GEM, a distributed goal evaluation algorithm for modern TM systems. Contrarily to most of the existing algorithms for TM, GEM detects when the computation has terminated (i.e., all the answers of a goal are collected) in a completely distributed way without the need of disclosing intensional policies, thereby preserving their confidentiality. GEM enables principals to delay the response to a request until a "maximal" set of answers have been computed, without running the risk of deadlocks. We demonstrate that the algorithm terminates and is sound and complete w.r.t. the standard semantics for logic programs. Even though GEM has been designed to perform distributed evaluation of TM policies, we argue that the proposed solution is general enough to be applied to other domains where mutual recursion and confidentiality of local programs are critical issues.

GEM can be easily extended to protect also the confidentiality of extensional policies. In particular, by not requiring intensional policies to be disclosed, the algorithm enables principals to discriminate between goals that may be accessed by other principals and goals that may only be used for internal computations, because of their sensitivity. This distinction is not possible when using a goal evaluation algorithm that relies on a centralized evaluation strategy (e.g., [9, 18]). A finer-grained protection of extensional policies can be achieved by employing trust negotiation (TN) algorithms [24, 25]. TN algorithms regulate the disclosure of extensional policies (i.e., possibly sensitive credentials) by means of rules that specify which credentials a requester must provide to get access to the requested credentials. GEM, on the other hand, provides a method for deriving these credentials. Therefore, GEM and TN algorithms are complementary, and can be combined to preserve the confidentiality of both intensional and extensional policies. More precisely, each GEM request might initiate a negotiation that, if successful, leads to the evaluation of the request.

The paper is structured as follows. Section 2 presents preliminaries on logic programming and SLD resolution. Section 3 introduces a basic version of GEM, whose implementation is presented in Section 4. Section 5 demonstrates the soundness, completeness and termination of the algorithm, and discusses what information is disclosed by GEM during the evaluation of a goal. Two possible extensions of GEM are presented in Section 6. Finally, Section 7 discusses related work and Section 8 concludes.

2. PRELIMINARIES

This section recaps the concepts of function-free logic programs [2] that are relevant to this paper.

An *atom* is an object of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms (i.e., variables and constants). An atom is *ground* if $t_1, \ldots, t_n$ are constants. A *clause* is a construct of the form $H \leftarrow B_1, \ldots, B_n$ (with $n \geq 0$), where H is an atom called *head* and $B_1, \ldots, B_n$ (called *body*) are atoms. If $n = 0$, the clause is a *fact*. A *program* is a finite set of clauses. We say that an atom $p(t_1, \ldots, t_n)$ is *defined in the program* P iff there is a clause in P that has $p(t_1, \ldots, t_n)$ in its head. A *goal* is a finite, possibly empty sequence of atoms $A_1, \ldots, A_n$. The empty goal is denoted by $\Box$.

SLD resolution (Selective Linear Definite clause resolution) [2] is the standard operational semantics for logic programs. In this paper, we consider SLD resolution with leftmost selection rule (extending the algorithm to an arbitrary selection rule is trivial). Computations are constructed as sequences of "basic" steps. Consider a goal $A_1, \ldots, A_n$ and a clause c in a program P. Let $H \leftarrow B_1, \ldots, B_m$ be a variant of c variable disjoint from $A_1, \ldots, A_n$. Let A_1 and H unify with *most general unifier* (*mgu*) θ. The goal $(B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$ is called a *resolvent of* $A_1, \ldots, A_n$ *and c with selected atom* A_1 *and mgu* θ. An SLD *derivation step* is denoted by $A_1, \ldots, A_n \xrightarrow{\theta} (B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$. Clause $H \leftarrow B_1, \ldots, B_m$ is called its *input clause*, and atom A_1 is called the *selected atom* of $A_1, \ldots, A_n$.

An SLD derivation is obtained by iterating derivation steps. The sequence $\delta := G_0 \xrightarrow{\theta_1} G_1 \xrightarrow{\theta_2} \cdots \xrightarrow{\theta_n} G_n \xrightarrow{\theta_{n+1}} \cdots$ is called a *derivation of* $P \cup \{G_0\}$, where at every step the input clause employed is variable disjoint from the initial goal G_0 and from the substitutions and the input clauses used at earlier steps. Given a program P and a goal G_0, SLD resolution builds a search tree for $P \cup \{G_0\}$, called *(derivation) tree* of G_0, whose branches are SLD derivations of $P \cup \{G_0\}$. Any selected atom in the SLD resolution of $P \cup \{G_0\}$ is called a *subgoal*.

SLD derivations can be finite or infinite. If $\delta := G_0 \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n} G_n$, with $\theta = \theta_1, \ldots, \theta_n$, is a finite prefix of a derivation, we say that δ is a *partial derivation* and θ is a *partial computed answer substitution* of $P \cup \{G_0\}$. If δ ends with the empty goal, θ is called *computed answer substitution* (*c.a.s.*). We also call θ a *solution* of G_0 and $G_0\theta$ an *answer* of G_0. The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ.

The most commonly employed mechanism to avoid infinite derivations is *tabling* [1, 4, 7, 11]. A goal G_0 defined in a program P is evaluated by producing a forest of (partial) derivation trees, one for each subgoal in the resolution of $P \cup \{G_0\}$. Each tree has an associated table, where the derived answers are stored. The evaluation of G_0 starts by ordinary resolution with the clauses in P: as in SLD, a subgoal G_1 is selected in a resolvent of G_0. If a tree for a variant of G_1 already exists, G_1 is added to the set of *consumers* of the corresponding table. Otherwise, a tree for G_1 is created. When a new answer of a subgoal is found, it is stored in the respective table and it is propagated to its consumer subgoals.

3. BASIC GEM

In this section we present a basic version of *GEM*, a goal evaluation algorithm for distributed (logic) programs. A distributed program is a collection of independent *local programs* stored at different locations. Differently from other TM languages (e.g., [1, 4]), we do not modify the standard notation of logic programming to represent the location where clauses are stored and evaluated. Instead, we assume that every atom has the form $p(loc, t_1, \ldots, t_n)$, where loc is a mandatory term representing the location of the program where the atom is defined, and $t_1, \ldots, t_n$ are terms. For instance, $p(bob, \ldots)$ refers to p as defined at Bob's location.

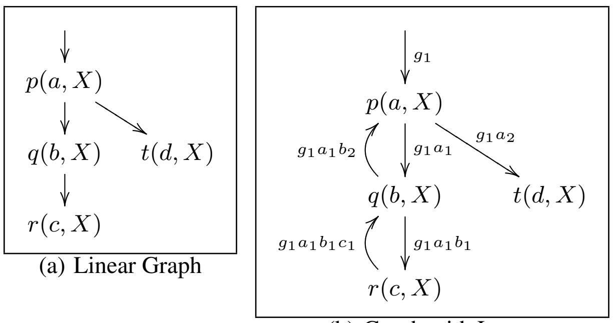
GEM requires the location parameter of an atom to be ground when the atom is selected for evaluation. If this is not the case, the computation *flounders*, as it is not possible to establish the local program in which the atom is defined. A discussion on how to write flounder-free programs and queries is orthogonal to the scope of this paper. Here, we just mention that there exist well-established techniques based on *modes* [3] which guarantee that certain parameters of an atom are ground when the atom is selected.

We assume a one-to-one correspondence between locations and principals: each principal is responsible for defining and evaluating the local program at her location (i.e., her policy). Consequently, each principal maintains the *partial derivation tree* of the atoms defined in the local program she controls.

DEFINITION 1. *Let G be a goal and P_G be the local program in which G is defined. A* partial derivation tree *of* G *is a derivation tree with the following properties:*

- *the root is the node* (G $\leftarrow$ G)*;*
- *there is a derivation step* (G $\leftarrow$ G) $\xrightarrow{\theta}$ $(G \leftarrow B_1, \ldots, B_n)\theta$*, where* (G $\leftarrow$ G) *is the root, iff there exists a clause $H \leftarrow B_1, \ldots, B_n$ in P_G (renamed so that it is variable disjoint from G) s.t. G and H unify with $\theta = mgu(G, H)$;*
- *let $(G \leftarrow B_1, \ldots, B_n)$ be a non-root node, and* Ans *be a set of answers of goal B_1; for each answer $B_1' \in$* Ans *(renamed so that it is variable disjoint from B_1) there is a derivation step $(G \leftarrow B_1, \ldots, B_n) \xrightarrow{\theta} (G \leftarrow B_2, \ldots, B_n)\theta$, where $\theta = mgu(B_1, B_1')$;*
- *for each branch* (G $\leftarrow$ G) $\xrightarrow{\theta_0}$ $(G \leftarrow B_1, \ldots, B_n)\theta_0 \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_n} (G \leftarrow \Box)\theta_0\theta_1 \ldots \theta_n$*, we say that $G\theta$ (with $\theta = \theta_0\theta_1 \ldots \theta_n$) is an answer of* G *using* clause $H \leftarrow B_1, \ldots, B_n$.

Since policy confidentiality is a major concern in TM systems, we assume that principals do not have access to the programs and the state of the computation at other principals' locations.



(a) Linear Graph

(b) Graph with Loops

Figure 1: Dependency Graphs of the Example Program

GEM performs a depth-first computation: a goal G is processed by evaluating one branch of its partial derivation tree at a time. This may involve the generation of evaluation requests for subgoals that are processed by different principals at different locations. When all the possible answers from each branch of the tree of G have been computed, they are sent to the requesters of G. G is *completely evaluated* when no more answers of G can be computed.

Suppose that a principal g sends to principal a a request for (the evaluation of) goal $p(a, X)$, which is evaluated w.r.t. the following distributed program:

```
1.  p(a,X) ← q(b,X).        4.  q(b,e).
2.  p(a,X) ← t(d,X).        5.  t(d,f).
3.  q(b,X) ← r(c,X).
```

Recall that the first parameter of the atom in the head indicates the principal evaluating the clause: the first two clauses are evaluated by principal a, clauses 3 and 4 by b, and clause 5 by d. Figure 1(a) shows the *dependency graph* of the program for initial goal $p(a, X)$. A dependency graph consists of a set of nodes and edges: nodes represent goals, and edges connect the head atom of a clause with the atoms in its body. In other words, edges correspond to (evaluation) requests.

When a receives the initial goal, she evaluates the first applicable clause in the local program (i.e., clause 1) and sends a request for $q(b, X)$ to b. In turn, b sends a request for $r(c, X)$ to c. c does not have any clause applicable to $r(c, X)$ and returns an empty set of answers to b. b evaluates the next applicable clause (i.e., $q(b, e)$), which is a fact. Since b does not have any other clause, she sends the computed answer to a. a applies the next clause (clause 2) and sends a request for $t(d, X)$ to d, who returns answer $t(d, f)$ to a after applying clause 5. At this point, $p(a, X)$ is completely evaluated and a sends answers $p(a, e), p(a, f)$ to g.

The evaluation of a subgoal of a goal G, however, may lead to new requests for G, forming a loop. Consider the program above with the following two additional clauses, stored by b and c respectively:

```
6.  q(b,X) ← p(a,X).
7.  r(c,X) ← q(b,X).
```

The new dependency graph is shown in Figure 1(b). Now, when c receives the request for $r(c, X)$, she applies clause 7 and sends a request for $q(b, X)$ to b, forming a loop. Similarly, the evaluation of clause 6 by b leads to another loop. However, a and b cannot detect whether a request forms a loop, as in a distributed system several independent requests for the same goal can occur.

In most of the existing goal evaluation systems (e.g., [7, 11, 18]), loop termination is made possible by the system's "global view" on the derivation process. Such global view, however, implies the loss of policy confidentiality. GEM detects loop termination in a completely distributed way without resorting to any centralized data structure. In particular, loops are handled in three steps: (1) detection, (2) processing, and (3) termination.

Loop Detection. Loops are detected by dynamically identifying Strongly Connected Components (SCCs), i.e., sets of mutually dependent subgoals. To enable the identification of SCCs, we assign to each request a unique identifier from an *identifier domain*.

DEFINITION 2. *An* identifier domain *is a triple $\langle I, \sqsubset, \hookrightarrow\rangle$, where:*

- *I is a set of strings of characters called* identifiers*;*
- *$\sqsubset$ is a partial order on the identifiers in I. Given two identifiers $id_1, id_2 \in I$ s.t. $id_1 \sqsubset id_2$, we say that id_1 is* lower *than id_2, and id_2 is* higher *than id_1;*
- *$\hookrightarrow$ is a partial order on the identifiers in I. Given two identifiers $id_1, id_2 \in I$ s.t. $id_1 \hookrightarrow id_2$, we say that id_2 is* side *of id_1.*

In addition, $\sqsubset$ and $\hookrightarrow$ have the following property: given the identifiers $id_1, id_2, id_3, id_4 \in I$ s.t. $id_1 \sqsubset id_2$, $id_3 \sqsubset id_4$, and $id_2 \hookrightarrow id_4$, then $id_1 \hookrightarrow id_3$.

Intuitively, $\sqsubset$ defines a top-down ordering, and $\hookrightarrow$ defines a left-right ordering w.r.t. the dependency graph of the distributed program. In other words, $\sqsubset$ reflects the order in which the goals in a branch of the graph are evaluated, whereas $\hookrightarrow$ reflects the order in which the branches are inspected. In the sequel, we consider identifiers obtained as follows: given a request for a goal G with identifier id_0, the identifier of the request for a subgoal G_1 in the partial derivation tree of G has the form $id_0 s_1$, denoting the concatenation of id_0 with a string of characters s_1. Then, $\sqsubset$ is a prefix relation: in the example, we have that $id_0 s_1 \sqsubset id_0$. Ordering $\hookrightarrow$ is a partial order on the strings composing the identifiers. For example, consider another subgoal G_2 of G with identifier $id_0 s_2$, which is evaluated after G_1. Then, we have that $id_0 s_1 \hookrightarrow id_0 s_2$.

In the dependency graph in Figure 1(b), edges are labeled with the corresponding request identifier. In particular, we concatenate the identifier of a request for a goal evaluated by principal a with meta-variables of the form a_i. Thus, for instance, a_1 and a_2 are distinct strings of characters generated by a.

A loop is detected when a request with identifier id_2 is received for a goal G s.t. a request id_1 for a variant of G has been previously received and $id_2 \sqsubset id_1$. Accordingly, we call the second request *lower request*, while the initial request for a goal is called *higher request*. Goal G is called the *coordinator* of the loop. We use the identifier of the higher request for G, id_1, as loop identifier.

An SCC may contain several loops. Given two loops with identifiers id_1 and id_2, we say that loop id_2 is *lower* than loop id_1 if $id_2 \sqsubset id_1$. The coordinator of the highest loop of the SCC (i.e., the loop with the highest identifier) is called the *leader* of the SCC.

In the example in Figure 1(b), identifiers g_1, g_1a_1, g_1a_2, and $g_1a_1b_1$ identify higher requests, while $g_1a_1b_1c_1$ and $g_1a_1b_2$ identify lower requests. Goals $q(b, X)$ and $p(a, X)$ are the coordinators of loops g_1a_1 and g_1 respectively. Loop g_1a_1 is lower than loop g_1, which is the highest loop of the SCC; therefore, $p(a, X)$ is the leader of the SCC. Looking at the identifier of the lower requests, a and b can determine the subgoals involved in the loop, which are $q(b, X)$ and $r(c, X)$ respectively. Goals inherit the ordering associated with the identifier of their higher request. In Figure 1(b), $p(a, X)$ is higher than $q(b, X)$ and $r(c, X)$.

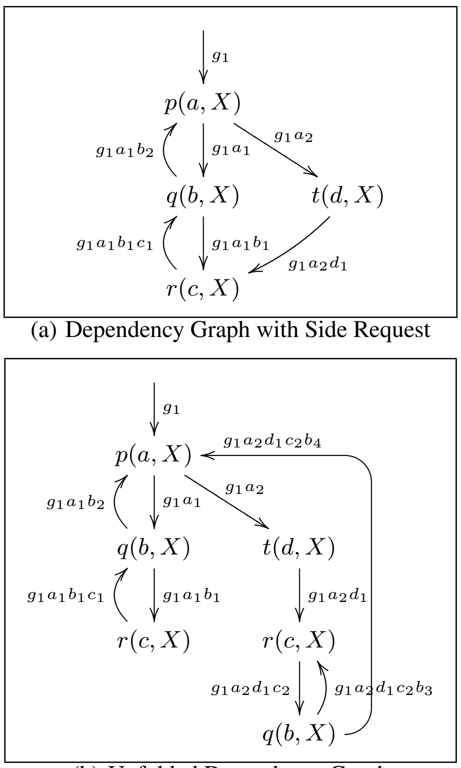
Loop Processing. As soon as a loop is detected, the answers of the coordinator already computed are sent to the requester of the lower request, together with a notification about the loop. The loop is then processed iteratively as follows: in turn, each principal (i) processes the received answers, (ii) "freezes" the evaluation of the subgoal involved in the loop, and (iii) evaluates other branches of the partial derivation tree of the locally defined goal. When all the branches have been evaluated, the new answers are sent to the requester of the higher request with a notification about the loop. When the notification gets back to the coordinator (together with a possibly empty set of new answers), all lower goals have been processed. If the processing of the received answers leads to new answers of the coordinator, these new answers are sent to the requesters of lower requests, starting a new iteration. Otherwise, a fixpoint has been reached (i.e., all possible answers of the goals in the loop have been computed) and the answers of the coordinator are sent to the requester of the higher request. Notice that a goal in a higher loop may eventually provide new answers to a goal in a lower loop: the fixpoint for a loop must be recalculated every time new answers of its coordinator are computed.

In Figure 1(b), when b identifies loop g_1a_1, she informs c that they both participate in loop g_1a_1. Since c has no more clauses to evaluate, she returns an empty set of answers to b notifying her that $r(c, X)$ is in loop g_1a_1. The further evaluation of $q(b, X)$ leads to the identification of loop g_1 and to a new answer $q(b, e)$, which is sent first to c in the context of loop g_1a_1. In turn, c computes answer $r(c, e)$ and sends it to b. Now, a fixpoint for loop g_1a_1 has been reached and b sends $q(b, e)$ to a notifying her that $q(b, X)$ is in loop g_1. Notice that $q(b, X)$ is also part of loop g_1a_1, but since this loop does not involve a, a is not notified of it. a computes answers $p(a, e)$ and $p(a, f)$ (the latter being found through the evaluation of $t(d, X)$), and sends them to b in the context of loop g_1. In turn, b computes $q(b, f)$. Now, b has to find a fixpoint for loop g_1a_1 given the new answer before proceeding with the evaluation of loop g_1. It is worth noting that c is not aware of loop g_1. This is because loop notifications are only transmitted to higher requests (except for the lower request that has formed the loop).

Loop Termination. The termination of the evaluation of all goals in the SCC is initiated by the principal handling the leader of the SCC. In the example, when the answers of $q(b, X)$ do not lead to new answers of $p(a, X)$, a informs b (which in turn informs c) that the evaluation of $p(a, X)$ is terminated and sends answers $p(a, e)$ and $p(a, f)$ to g.

Complex Loops. So far we have only considered *simple* loops, i.e., loops formed by lower requests. More complex loops are formed when the evaluation of a goal G in a SCC leads to a higher request for a goal G_1 in the same SCC. In other words, complex loops occur when a request with identifier id_2 is received for G_1 s.t. a request id_1 for a variant of G_1 has been previously received and $id_1 \hookrightarrow id_2$. Accordingly, we refer to this type of requests as *side requests*, and we call G a *side goal.*

The main problem with side requests is that it is difficult to determine when they should be responded to. Indeed, even though it is clear that the two goals involved in the side request participate in at least one common loop (if this was not the case G_1 would be disposed by the time the side request is received), it may not be possible to identify (all) the common loops. Technically, to enable the detection of termination, a side request should be responded only when a fixpoint is computed for all the loops lower than the loop in which the side goal is involved. Consider, for instance, the



(a) Dependency Graph with Side Request

(b) Unfolded Dependency Graph

Figure 2: Dependency Graphs with Side Requests

following additional clause stored by d:

```
8.  t(d,X) ← r(c,X).
```

The new dependency graph is shown in Figure 2(a). Now, if c sends answers to d at every iteration of loop g_1a_1, a would not know when to stop waiting for answers from d (since a does not know on which goals $t(d, X)$ depends). On the other hand, c cannot wait until a fixpoint is reached for loop g_1a_1, since only b (the principal handling coordinator) is aware of that.

The simplest solution to this problem, which we present in Section 4, is to treat a side request as a new request and to reevaluate the goal. Accordingly, when a side request is received, the algorithm creates a new partial derivation tree for the goal. This corresponds to inspecting multiple times some branches of the dependency graph of the program (Figure 2(b)). However, this solution is not satisfactory in practice because the size of the unfolded graph is in the worst case exponential w.r.t. the number of nodes in the original graph. In Section 6 we discuss an alternative solution that preserves soundness, completeness, and termination without the need of reevaluating side requests.

4. IMPLEMENTATION

In this section we introduce the data structures and procedures used by basic GEM to evaluate a goal.

Data Structures. In GEM, principals communicate by exchanging *request* and *response* messages.

DEFINITION 3. *A* request *is a triple $\langle id, req, \mathrm{G}\rangle$, where:*

- *id is the request identifier;*
- *req is a principal, called* requester*;*
- G *is a goal $p(loc, t_1, \ldots, t_n)$ (where loc is a constant).*

A *request* is an enquiry issued by principal req for the evaluation of goal G. Each *request* is uniquely identified by id and is sent to the principal defining G.

DEFINITION 4. *Let $r = \langle id, req, \mathrm{G}\rangle$ be a request. A* response *to r is a tuple $\langle id, Ans, S_{ans}, Loops\rangle$, where:*

- *id is the response identifier;*
- *Ans is a (possibly empty) set of answers of* G*;*
- *$S_{ans} \in \{active, loop(id'), disposed\}$ is the status of the evaluation of* G*, where id' is a loop identifier;*
- *Loops is a set of loop identifiers.*

A response has the same identifier of the request to which it refers. It contains a (possibly empty) set of answers of the requested goal (Ans), together with the status of its evaluation and information about the loops in which it is involved ($Loops$). The status is $disposed$ if the goal has been completely evaluated, $active$ if additional response messages are expected, and $loop(id')$ if the *response* is sent in the context of the evaluation of loop id'. Intuitively, id' identifies the coordinator waiting for the answers.

GEM computes the answers of a goal by a depth-first evaluation of its partial derivation tree. We represent a partial derivation tree as a structure called *partial tree*. The building blocks of a partial tree are called *nodes*.

DEFINITION 5. *A* node *is a triple $\langle id, c, S\rangle$, where:*

- *id is the node identifier;*
- *c is a program clause;*
- *$S \in \{new, active, loop(ID), answer, disposed\}$ is the status of the evaluation of the selected atom in c, where ID is a set of loop identifiers.*

The status of a node is new if no atom from the body of c has yet been selected for evaluation, $active$ when a body atom is being evaluated, and $disposed$ when the selected atom is completely evaluated. The status is *loop(ID)* if the selected atom is part of some loops, and $answer$ if c is a fact.

DEFINITION 6. *The* partial tree *of a goal* G *is a tree with the following properties:*

- *the root is node $\langle id_0, \mathrm{G} \leftarrow \mathrm{G}, S\rangle$;*
- *there is an edge from the root to a node $\langle id_1, (G \leftarrow B_1, \ldots, B_n)\theta, S'\rangle$, where $id_1 \sqsubset id_0$, iff there exists a derivation step $(G \leftarrow G) \overset{\theta}{\to} (G \leftarrow B_1, \ldots, B_n)\theta$ in the partial derivation tree of* G*;*
- *there is an edge from node $\langle id_2, G \leftarrow B_1, \ldots, B_n, S''\rangle$ to node $\langle id_3, (G \leftarrow B_2, \ldots, B_n)\theta, S'''\rangle$, where $id_2 \sqsubset id_0$ and $id_3 \sqsubset id_0$, iff there exists a derivation step $(G \leftarrow B_1, \ldots, B_n) \overset{\theta}{\to} (G \leftarrow B_2, \ldots, B_n)\theta$ in the partial derivation tree of* G*;*

When a principal receives a higher request for a goal G, it creates a table for G. A table contains all the information about the evaluation of G.

DEFINITION 7. *The* table *of a goal G, denoted $Table(G)$, is a tuple $\langle HR, LR, ActiveGoals, AnsSet, Tree\rangle$, where:*

- *HR is a higher request for G;*
- *LR is a set of lower requests for G;*

- *ActiveGoals is a set of pairs $\langle id, counter\rangle$ where id is a loop identifier and $counter$ is an integer value;*
- *$AnsSet$ is a set of pairs $\langle ans, ID\rangle$ where ans is an answer of G and ID is a set of request identifiers;*
- *$Tree$ is the partial tree of G.*

The table of a goal G stores the higher request *HR* for which it has been created, the set of answers computed so far ($AnsSet$), and the partial tree of G ($Tree$). Possible lower requests for G are stored in LR. *ActiveGoals* keeps a counter for each loop in which G is involved; the counter of a loop id indicates the number of subgoals in *Tree* which are involved in loop id, and is decreased whenever an answer of one of these subgoals is received. The status of the root node of *Tree* indicates the status of the evaluation of G.

Procedures. To initiate the evaluation of a new goal G, a principal a generates a random identifier id and sends the request $\langle id, a, G\rangle$ to the principal defining G.[1] A response $\langle id, Ans, disposed, \{\}\rangle$ is returned to a when the algorithm terminates. GEM computes the answers of G (defined in the local program P_G) by means of the following procedures:

- CREATE TABLE: if the request is not a lower request, creates a table for G and initializes its partial tree with the applicable clauses in P_G;
- ACTIVATE NODE: activates a new node in the partial tree of G;
- PROCESS RESPONSE: processes the answers received for a subgoal in the partial tree of G;
- GENERATE RESPONSE: determines the requesters of G to which a response must be sent. It is invoked when there are no more nodes to activate;
- SEND RESPONSE: sends the computed answers to the requesters of G;
- TERMINATE: disposes the table of G. It is invoked when G is completely evaluated.

Each principal in the distributed system runs a listener which waits for incoming requests and responses. Whenever a new request is received, the listener invokes CREATE TABLE. Similarly, PROCESS RESPONSE is invoked upon receiving a response to a previously issued request.

CREATE TABLE (Algorithm 1) inputs a request $\langle id_0, req, G\rangle$ and, if there exists no table for a variant of G, it creates a table for G with *HR* set to $\langle id_0, req, G\rangle$, and *Tree* initialized with the clauses in the local program applicable to G (renamed so that they share no variable with G) (lines 19-25). The identifiers of the subnodes of the root are obtained by concatenating id_0 with a randomly generated string of characters s. When the initialization of the table of G is terminated, ACTIVATE NODE is invoked (line 26).

If another request for goal G (or a variant of G) has been previously received, three situations are possible:

1. *The request refers to a goal which has been completely evaluated* (lines 3-4). In this case, a response with all the answers of G is sent to the requester by invoking SEND RESPONSE.
2. *The request is a lower request for G* (lines 5-7). This corresponds to the detection of loop id_1 (where id_1 is the identifier of *HR*). The request is added to the set of lower requests *LR* and the answers computed so far are sent to the requester, together with a notification about loop id_1.

[1]Here, we assume that standard cryptographic techniques are in place to avoid collision of identifiers.

Algorithm 1: CREATE TABLE

```
input: a request $\langle id_0, req, G\rangle$
1  if $\exists Table(G') = \langle\langle id_1, req', G'\rangle, LR, AG, AS, T\rangle$ s.t. $G'$ is a variant of $G$ then
2      let $S_{root}$ be the status of the root node of $T$
3      if $S_{root} = disposed$ then
4          SEND RESPONSE($\langle id_0, req, G'\rangle, disposed, \{\}$)
5      else if $id_0 \sqsubset id_1$ then
6          $LR := LR \cup \{\langle id_0, req, G'\rangle\}$
7          SEND RESPONSE($\langle id_0, req, G'\rangle, active, \{id_1\}$)
8      else
9          let $G''$ be a variable renaming of $G$ s.t. $\nexists Table(G'')$
10         create $Table(G'')$
11         initialize $Table(G'')$ to
           $\langle\langle id_0, req, G''\rangle, \{\}, \{\}, \{\}, \langle id_0, G'' \leftarrow G'', new\rangle\rangle$
12         foreach clause $H \leftarrow B_1, \ldots, B_n$ applicable to $G$ in the local program do
13             let $H' \leftarrow B_1', \ldots, B_n'$ be a variable renaming of the clause s.t. it is variable disjoint from $G''$, and $\theta = mgu(G'', H')$
14             let $s$ be a randomly generated string of characters
15             add subnode $\langle id_0 s, (H' \leftarrow B_1', \ldots, B_n')\theta, new\rangle$ to the root
16         end
17         ACTIVATE NODE($G$)
18 else
19     create $Table(G)$
20     initialize $Table(G)$ to
       $\langle\langle id_0, req, G\rangle, \{\}, \{\}, \{\}, \langle id_0, G \leftarrow G, new\rangle\rangle$
21     foreach clause $H \leftarrow B_1, \ldots, B_n$ applicable to $G$ in the local program do
22         let $H' \leftarrow B_1', \ldots, B_n'$ be a variable renaming of the clause s.t. it is variable disjoint from $G$, and $\theta = mgu(G, H')$
23         let $s$ be a randomly generated string of characters
24         add subnode $\langle id_0 s, (H' \leftarrow B_1', \ldots, B_n')\theta, new\rangle$ to the root
25     end
26     ACTIVATE NODE($G$)
```

3. *The request is a side request or originates from a different initial request* (lines 8-17). We treat the request as a new request: a new table for G is created and ACTIVATE NODE is invoked.

ACTIVATE NODE (Algorithm 2) activates a new node from the partial tree of a goal G. First, a node with status new is selected from T (line 5). If the node's clause is a fact and represents a new answer, it is added to the set of answers AS (with an empty set of recipients), and ACTIVATE NODE is invoked again (lines 6-10). Otherwise, the leftmost atom B_1 of the body of the clause is selected for evaluation. In case that the location parameter of B_1 is not ground, an error is raised and the computation is aborted by *floundering* (lines 12-13). Otherwise, a request for B_1 is sent to the corresponding location; the node identifier is used as request identifier (lines 15-19). If there are no more nodes with status new, or G is in the set of computed answers AS, GENERATE RESPONSE is invoked (lines 2-3).

SEND RESPONSE (Algorithm 3) inputs a request, a response status, and a set of loop identifiers and sends a response message to the requester, which includes the answers of G that have not been previously sent to that requester.

Responses are processed by PROCESS RESPONSE (Algorithm 4). The node n to which the response refers is identified by looking at the response identifier (line 1). If the status of the response

Algorithm 2: ACTIVATE NODE

```
input: a goal $G$
let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
if ($\nexists$ a non-root node $n \in T$ with status new) or ($\langle G, ID\rangle \in AS$) then
    GENERATE RESPONSE($G$)
else
    select a non-root node $n = \langle id_1, H \leftarrow B_1, \ldots, B_n, new\rangle$ from $T$
    if $n = 0$ then
        set the status of $n$ to $answer$
        if $H$ is not subsumed by any answer in $AS$ then
            $AS := AS \cup \{\langle H, \{\}\rangle\}$
        ACTIVATE NODE($G$)
    else
        if the location of $B_1$ is not ground then
            halt with an error message  /* floundering */
        else
            set the status of $n$ to $active$
            let $S_{root}$ be the status of the root node of $T$
            if $S_{root} = new$ then
                $S_{root} := active$
            send request $\langle id_1, local, B_1\rangle$ to the location of $B_1$
```

Algorithm 3: SEND RESPONSE

```
input: a request $\langle id_0, req, G\rangle$, a response status $S_{ans}$, a set of
       loop identifiers $Loops$
let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
$Ans := \{\}$
foreach $\langle ans, ID\rangle \in AS$ s.t. $id_0 \notin ID$ do
    $Ans := Ans \cup \{ans\}$
    $ID := ID \cup \{id_0\}$
end
send response $\langle id_0, Ans, S_{ans}, Loops\rangle$ to $Req$
```

is *disposed*, the selected atom B_1 of n is completely evaluated. Therefore, n is disposed and, if B_1 is in a loop, also all the other nodes in any loop of the SCC are disposed (lines 5-8). This is because the termination of a loop is ordered by the principal handling the leader of the SCC once all the goals (and consequently, all the loops) in the SCC are completely evaluated.

Otherwise, the status of n is updated depending on whether the response contains a loop notification, i.e., *Loops* contains some loop identifier (lines 10-13). In this case, an entry is added to the set of active goals AG for each new loop in *Loops* (line 14). If the response has been sent in the context of the evaluation of a loop id_3, the counter of id_3 in AG is decreased and the status of the table is changed to $loop(\{id_3\})$ (lines 15-18).

After updating the node and table status, the set of answers in the response is processed (lines 19-23). In particular, a new subnode of n is created for each answer. The clause of the new node is $(H \leftarrow B_2, \ldots, B_n)\theta$, where θ is the mgu of B_1 and the answer, and its identifier is obtained by concatenating the identifier id_1 of the root node of T with a randomly generated string of characters s. When all answers have been processed, if the principal is not waiting for a response for any subgoal in the partial tree of G, ACTIVATE NODE is invoked to proceed with the evaluation of G (line 25).

GENERATE RESPONSE (Algorithm 5) is invoked when all the clauses in the partial tree of a goal G (except for the ones in a loop) have been evaluated. If G is not part of a loop, TERMINATE is invoked (lines 2-3). Otherwise, we distinguish three cases:

Algorithm 4: PROCESS RESPONSE

```
input: a response $\langle id_0, Ans, S_{ans}, Loops\rangle$
let $n = \langle id_0, H \leftarrow B_1, \ldots, B_n, S_n\rangle$ be the node in the partial
    tree of goal $G$ to which the response refers
let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
let $\langle id_1, G \leftarrow G, S_{root}\rangle$ be the root node of $T$
if $S_{root} \neq disposed$ then
    if $S_{ans} = disposed$ then
        if $S_n = loop(ID)$ then
            dispose all the nodes in $T$ involved in any loop
        $S_n := disposed$
    else
        if $S_n = loop(ID)$ then
            $ID := ID \cup Loops$
        else if $Loops \neq \{\}$ then
            $S_n := loop(Loops)$
        $AG := AG \cup \{\langle id_2, 0\rangle | id_2 \in Loops$ and $\langle id_2, c\rangle \notin AG\}$
        if $S_{ans} = loop(id_3)$ then
            decrease the counter of $id_3$ in $AG$ by 1
            if $S_{root} = active$ then
                $S_{root} := loop(\{id_3\})$
    foreach $ans \in Ans$ do
        let $ans'$ be a variable renaming of $ans$ s.t. it is variable
        disjoint from $B_1$, and $\theta = mgu(B_1, ans')$
        let $s$ be a randomly generated string of characters
        add subnode $\langle id_1 s, (H \leftarrow B_2, \ldots, B_n)\theta, new\rangle$ of $n$
    end
    if ($S_{root} = active$) or ($S_{root} = loop(ID)$ and $\forall id_4 \in ID$,
    $\langle id_4, 0\rangle \in AG$) then
        ACTIVATE NODE($G$)
```

Algorithm 5: GENERATE RESPONSE

```
input: a goal $G$
let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
if ($\nexists\langle id_0, c, loop(ID)\rangle \in T$) or ($\langle G, ID'\rangle \in AS$) then
    TERMINATE($G$)
else
    let $\langle id_1, G \leftarrow G, S_{root}\rangle$ be the root node of $T$
    if $G$ is the coordinator of a loop $id_1$ and $\exists ans \in AS$ s.t. $ans$
    has not been sent to some request in $LR$ then
        set the counter of $id_1$ in $AG$ to the number of subgoals in $T$
        involved in loop $id_1$
        if $S_{root} = loop(ID'')$ then
            $S_{root} := loop(ID'' \cup \{id_1\})$
        else
            $S_{root} := loop(\{id_1\})$
        foreach $\langle id_2, req, G\rangle \in LR$ do
            SEND RESPONSE($\langle id_2, req, G\rangle, loop(id_1), \{\}$)
        end
    else if $G$ is the leader of the SCC then
        TERMINATE($G$)
    else
        let $Loops$ be the set $\{id_3 | \langle id_3, C\rangle \in AG$ and $id_1 \sqsubset id_3\}$
        set the counter of each $id_3 \in Loops$ to the number of
        subgoals in $T$ in loop $id_3$
        if $S_{root} = loop(ID'')$ and $\exists id_4 \in ID''$ s.t. $id_1 \sqsubset id_4$ then
            SEND RESPONSE($HR, loop(id_4), Loops$)
        else
            SEND RESPONSE($HR, active, Loops$)
        $S_{root} := active$
```

1. If G is the coordinator of a loop id_2 and there are new answers to be sent to the lower requests in LR, a response with status $loop(id_2)$ is sent to each of them (lines 6-14). The status of the root node of T keeps track of the loops that are

Algorithm 6: TERMINATE

```
input: a goal $G$
1 let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
2 dispose all non-answer nodes in $T$
3 foreach $\langle id_0, \text{req}, G\rangle \in \{HR\} \cup LR$ do
4     SEND RESPONSE($\langle id_0, req, G\rangle, disposed, \{\}$)
5 end
6 $HR := LR := AG := \{\}$
```

currently being processed and the counter of id_2 in AG is set to the number of subgoals in T involved in loop id_2 (i.e., the number of subgoals for which a response in the context of loop id_2 will be returned).

2. If G is the leader of the SCC and no new answers of G have been computed, the loop is terminated by invoking TERMINATE (lines 15-16).
3. Otherwise, a response including the identifier of the loops in which G is involved is sent to the requester of *HR* (lines 18-24). The status of the response depends on whether *HR* is involved in one of the loops that are currently being processed.

TERMINATE (Algorithm 6) is responsible of disposing a table once all the answers of its goal G have been computed. A response with status $disposed$ is sent to the requesters of *HR* and *LR* (lines 3-5).

5. PROPERTIES OF GEM

In this section we present the soundness, completeness and termination results of basic GEM. Furthermore, we discuss what information is disclosed by GEM during the evaluation of a goal.

Soundness, Completeness and Termination. Here, we refer to an arbitrary but fixed set $P_1, \ldots, P_n$ of local programs, and to the corresponding distributed program $P = P_1 \cup \ldots \cup P_n$. To prove soundness and completeness, we demonstrate that GEM computes a solution iff such a solution can be derived via SLD resolution, which has been proved sound and complete [2].

THEOREM 1 (SOUNDNESS). *Let G_1 be a goal. Let S be the set of tables resulting from running GEM on G_1 (w.r.t. $P_1, \ldots, P_n$). Let $G_1, \ldots, G_k$ be the goals for which there exists a table in S. For each goal $G_i \in \{G_1, \ldots, G_k\}$ let $Sol_i = \{\theta_{i,1}, \ldots, \theta_{i,k_i}\}$ be the (possibly empty) set of solutions of G_i generated by the algorithm. Then, for each $G_i \in \{G_1, \ldots, G_k\}$ and for each $\theta_{i,j} \in Sol_i$ there exists an SLD derivation of $P \cup \{G_i\}$ with c.a.s. σ s.t. $G_i\theta_{i,j}$ is a renaming of $G_i\sigma$.*

THEOREM 2 (COMPLETENESS). *Let G_1 be a goal. Let S be the set of tables resulting by running GEM on G_1 (w.r.t. $P_1, \ldots, P_n$). Assume that running GEM on G_1 (w.r.t. $P_1, \ldots, P_n$) did not result in floundering. If there exists an SLD derivation of $P \cup \{G_1\}$ with c.a.s. θ, then there exists a solution σ of G_1 in S s.t. $G_1\theta$ is a renaming of $G_1\sigma$.*

For termination, we have to prove the following theorem.

THEOREM 3 (TERMINATION). *Given a goal G evaluated w.r.t. a finite distributed program P, GEM terminates.*

Termination is a consequence of two observations: (i) the dependency graph of P is finite, and (ii) the number of response messages exchanged by the principals involved in the evaluation of G is finite. Proofs of soundness, completeness and termination are given in Appendix A.

Disclosed Information. A primary objective of GEM is to preserve the confidentiality of local programs (i.e., intensional policies). Here, we discuss what policy information principals are able to collect during the evaluation of a goal. We say that a goal G_1 *depends* on a goal G_2 if there is a path (i.e., a chain of requests) from G_1 to G_2 in the dependency graph of the distributed program in which the goals are defined.

Let a be a principal, and G_1 be any goal in a distributed program. The question that we address is what can a infer about the definition of G_1. Obviously, by requesting the evaluation of G_1, a can learn the set of answers to the request, i.e., the *extensional* policy relative to G_1; this is necessary for any goal evaluation algorithm. However, we are more interested in what a can learn about the *intensional* policy relative to G_1.

First of all, by sending a request for G_1 (say, with identifier id_1), a can learn whether G_1 depends on a predicate defined in her policy. Indeed, if a receives a request for a goal G_2 with identifier id_2 s.t. $id_2 \sqsubset id_1$, then G_1 depends on G_2.

If G_1 depends on a number of goals defined by a, then by requesting G_1 a learns:

- What are the goals $G_2, \ldots, G_n$ defined by a that G_1 depends on.
- For each $G_i \in \{G_2, \ldots, G_n\}$, a learns the principal p_i that requested G_i; therefore, a learns that G_1 depends on some predicate defined by p_i. Principal a, however, does not learn which predicate it is.
- Depending on the implementation of identifiers, a might be able to learn also some information about the length of the derivation chain from G_1 to G_i. For example, if identifiers were constructed by concatenating the identifier of a request with strings of fixed length, a would be able to infer the number of goals involved in the chain of requests from G_1 to G_i.

We believe that the information disclosed by GEM is not only necessary to handle loops correctly, but also consistent with the concept of TM. A principal a is only able to infer if and how a given goal G_1 depends on predicates defined by herself. If this is the case, then there is a *chain of trust* from the principal defining G_1 to a; it seems legitimate that the existence of this chain of trust should not remain secret to a. We also argue that the knowledge about goal dependencies disclosed by GEM is not sufficient for a to infer the intensional policy of the other principals. Indeed, different intensional policies can have the same dependency graph [8].

6. EXTENSIONS OF BASIC GEM

This section discusses two possible extensions of basic GEM, one for avoiding the reevaluation of side requests and the other to deal with negation (as failure).

GEM + Early Loop Detection. In basic GEM, side requests need to be reevaluated for two reasons: (1) principals are not aware of all the loops in which they are involved (loop notifications are only transmitted to higher requests), and (2) only the principal handling a coordinator knows when the fixpoint for the loop is reached. This implies that principals might not be able to know when a response to side requests should be sent.

In this section, we propose a variant of GEM which avoids the reevaluation of side requests by enabling *early loop detection.* In practice, we add a *loop finalization* step each time a fixpoint for a loop is reached that enables principals to identify the loops in which a side request is involved and to determine when it should be responded to. When a fixpoint for a loop is reached, the principal handling its coordinator (say goal G) sends a *finalization message* to the requesters of lower requests for G, who in turn propagate the message to the other principals involved in the loop, similarly to a loop notification. During this propagation process, principals that handle coordinators of lower loops forward the message to the principals involved in such loops, so that eventually all the principals handling goals lower than G are informed.

The finalization message that the principal handling the coordinator of a loop id_c transmits to the other principals in its loop is a tuple $\langle id_r, id_c, id_f, id_l, ID\rangle$, where id_r identifies the request in response to which the message is sent, id_c indicates the "source" of the finalization message, id_f is the loop for which a fixpoint has been reached, id_l identifies the loop whose coordinator is waiting for a response, and set ID contains the identifiers of all the loops in which the coordinator of loop id_c is involved. Notice that no finalization is needed for the highest loop of the SCC: when a fixpoint for that loop is reached, all the goals in the SCC (including side goals) have been completely evaluated.

By knowing *all* the loops to which their subgoals participate (ID), principals can detect the loops in which a side request with identifier id_1 is involved. This is done by checking if there is any loop identifier id known by the principal s.t. $id_1 \sqsubset id$. A principal responds to the side request with a notification about the loops in which it is involved. If there is no known loop with identifier id s.t. $id_1 \sqsubset id$, the request is not a side request, but originates from a different initial request and the requested goal is reevaluated. The reevaluation of a goal for each different initial request is necessary to identify SCCs.

Identifier id_c allows a principal to know *when* a response should be sent to side requests. More precisely, id_c is used to determine how many finalization messages will be received on behalf of the same coordinator. When all such messages have been received (i.e., the counter of id_c is 0), a response can be sent to the side requests and the finalization message is propagated to the higher requests. This guarantees that all the subgoals of the locally evaluated goal have been processed when a response is sent to higher or side requests.

By knowing the loop for which a fixpoint is reached (id_f), a principal can identify *to which* side requests a response should be sent. More specifically, the principal has to respond to the side requests involved in the "next" loop, i.e., the loop whose identifier has the longest prefix w.r.t. id_f among the known loops. Identifier id_l determines the status of the response. This information is necessary to guarantee the correct functioning of counters.

In the implementation, we introduce a new field SR (the set of Side Requests) to the table of a goal, and *ActiveGoals* is now a list ordered by increasing identifier (i.e., from lower to higher loops). Upon receiving a finalization message, a principal first identifies the goal and the table to which the message refers, by looking at id_r. Then, the principal adds set ID to *ActiveGoals* in the appropriate order, and decreases the counter of id_c. If the counter of id_c is 0, the message is propagated to *HR* (if it is involved in loop id_f), and a response with status $loop(id_l)$ is sent to the side requests in SR involved in the loop following id_f in *ActiveGoals*.

The soundness, completeness, and termination results of this variant of GEM are discussed in Appendix A.

Dealing with Negation. Basic GEM is devised to work with *definite* logic programs, i.e., programs without negation. Negation is used by some TM systems (e.g., [10, 12]) to express non-monotonic constraints in the body of a clause, such as separation of duty or "distrust" in principals with certain attributes (e.g., employees of a rival company). Here, we propose a relatively simple extension of GEM to support the use of negation. In particular, we extend the algorithm as follows. Given a clause with a literal $\neg B$ selected for evaluation:

1. if B is not ground, an error is raised and the computation flounders;
2. if the evaluation of atom B succeeds with an answer, then $\neg B$ fails and the clause is disposed;
3. if B is completely evaluated and has no answers, then $\neg B$ succeeds and a new clause is added to the partial tree of the goal, removing $\neg B$ from the body;
4. if a loop notification for atom B is received, an error is raised and the computation flounders.

Conditions (1), (2) and (3) are standard when defining *negation as failure*: (1) is necessary to guarantee correctness [2], while (2) and (3) define the semantics of negation. Since GEM is able to detect loops, (3) captures also the case of infinite failure, as done, for instance, by the well-founded semantics [13]. Condition (4) states that the algorithm flounders when it detects a *loop through negation* (e.g., for clause $p \leftarrow \neg p$); this is because dealing consistently with loops through negation requires the use of a three-valued semantics [21]. Relying on three-valued models in the context of access control, however, raises the issue of how to interpret the "undefined" answer, as the result of a computation must always be mapped to a positive or negative decision. It is straightforward to demonstrate that the proposed extension of GEM:

- always terminates (for arbitrary programs and requests), and
- for non-floundering computations, it is sound and complete w.r.t. the Well-Founded semantics [13].

It is also worth noting that both types of floundering can be avoided by imposing restrictions on the program. In particular, floundering of type (1) can be avoided by restricting to *well-moded* programs [3], while floundering of type (4) can be avoided by restricting to *weakly stratified* programs [20], which include *locally stratified* and *stratified* programs [19]. However, while the definition of well-moded program requires each clause *independently* to be well-moded, the definition of weakly stratified program relies on a "global ordering" among *all* the (ground) atoms in a program. In the context of TM, where a program is a collection of independent local programs defined by different principals, enforcing well-modeness is feasible, but guaranteeing that a program is weakly stratified requires principals to agree beforehand on the allowed dependencies among (ground) goals. The added value of GEM is that it does not require a program to be weakly stratified: its loop identification mechanism detects loops through negation at run-time, preserving soundness, completeness and termination of the algorithm for non-floundering programs and requests.

7. RELATED WORK

Research on goal evaluation has been carried out both in the field of logic programming (LP) and TM. In this section we compare our work with existing frameworks focusing on the information disclosed during the evaluation process. Additionally, we indicate whether the analyzed systems employ a centralized or distributed goal evaluation algorithm and termination detection mechanism. Within termination detection, we distinguish between termination of the whole computation initiated by a particular request and termination of the single goals involved in the computation (i.e., detecting when a goal is completely evaluated). Table 1 summarizes the results of this analysis.

Frameworks		Evaluation	Computation Termination	Goal Termination	Disclosed Information
LP	SLG [7]	centralized	centralized	centralized	intensional policy, extensional policy
	YapTab [22]	centralized	centralized	centralized	intensional policy, extensional policy
	Hulin [15]	centralized	centralized	centralized	intensional policy, extensional policy
	Damasio [11]	distributed	distributed	distributed	dependency graph, extensional policy
	Hu [14]	distributed	distributed	distributed	dependency graph, extensional policy
TM	RT [18]	centralized	centralized	centralized	intensional policy, extensional policy
	Tulip [9]	centralized	centralized	centralized	intensional policy, extensional policy
	SD3 [16]	distributed	N/A	N/A	intensional policy, extensional policy
	Becker et al. [5]	distributed	N/A	N/A	intensional policy, extensional policy
	Cassandra [4]	distributed	no	no	extensional policy
	PeerTrust [1]	distributed	distributed	no	extensional policy
		distributed	distributed	distributed	dependency graph, extensional policy
	MTN [26]	distributed	distributed	no	extensional policy
	GEM	distributed	distributed	distributed	some goal dependencies, extensional policy

Table 1: Summary of Related Work

SLG [7], Yaptab [22] and the work by Hulin [15] are centralized tabling systems in which the complete program is available during the evaluation. SLG identifies loops by observing goal dependencies in the "call stack" of the program, and termination is detected when no more operations can be applied to the goals in the stack. Yaptab and Hulin focus on improving the efficiency of goal evaluation by proposing a parallel tabled execution strategy. Similarly to SLG, Yaptab resorts to centralized data structures to identify loops and detect termination. In [15], each process communicates its termination to a global variable, whose access is limited to one process at a time by means of a deadlock mechanism.

Distributed goal evaluation frameworks are presented in [11] and [14]. In [11], termination detection resorts to a static dependency graph known to all principals and determined at compile time. The work by Hu [14] assumes the presence of global data structures, and requires goal dependencies to be propagated among the different principals. Consequently, the confidentiality of (part of) intensional policies is not preserved.

In TM, distributed goal evaluation is a main issue as policies are distributed among principals. RT [18] and Tulip [9] rely on a centralized goal evaluation strategy: all the clauses necessary for the evaluation of a goal are collected in a single location. In SD3 [16], when queried for a goal, a principal returns to the requester the clauses defining the goal, with locally defined atoms already evaluated. Becker et al. [5] present an algorithm in which the body atoms of a clause are sent in turn to the principals responsible for their evaluation; each principal evaluates the locally defined atom(s) and sends all the atoms to the next principal. Neither [16] nor [5] discuss how termination is detected. Cassandra [4] employs a completely distributed evaluation strategy in which no information about intensional policies is disclosed among principals. However, it does not detect neither the complete evaluation of single goals, nor the termination of the whole computation.

PeerTrust [1] and MTN [26] detect termination of the computation started by a particular request in a fully distributed way; this is achieved by "observing" when no more messages are exchanged among principals and all the goals are quiescent. In [1], the authors present two solutions: the first, based on [11], is also able to detect the completion of single goals, but requires the dependency graph of the distributed program to be known to all principals a priori. The second solution, which is also adopted in [26], detects termination of the computation without disclosing information about intensional policies. However, since all messages are tagged with the identifier of the initial request, some information about goal dependencies could be inferred as discussed in Section 5. In addition, this solution does not detect termination of individual goals, which is required to free the resources used during evaluation and to allow the use of negation. Finally, neither PeerTrust nor MTN provide a loop identification mechanism; when using negation, the detection of possible loops through negation allows to preserve the soundness and completeness of the computation w.r.t. the standard semantics of logic programs. We enable the detection of goal termination and the identification of loops at the cost of possibly revealing (at run time) some goal dependencies. In Section 6 we have shown how GEM can be extended to support a restricted use of negation.

8. CONCLUSIONS

In this paper we have presented GEM, a distributed goal evaluation algorithm based on a foundation language for modern TM systems. GEM detects termination of the computation in a completely distributed way without the need of disclosing intensional policies, thereby preserving their confidentiality. We have proved that the algorithm terminates and is sound and complete w.r.t. the standard semantics for logic programs. As future work, we plan to extend GEM to support constraint rules [17].

In a TM system, confidentiality of extensional policies is also an important requirement, as the answers of a goal might contain sensitive information (e.g., the list of patients of a mental hospital). Therefore, the disclosure of extensional policies should also be protected. Confidentiality of both intensional and extensional policies can be preserved by complementing GEM with a trust negotiation algorithm [24, 25]. In particular, each GEM request might initiate a negotiation that, if successful, leads to the evaluation of the request.

Although efficiency is not a primary objective of this paper, GEM can contribute to keep network traffic low. In most distributed goal evaluation systems, answers are sent as soon as they are computed. On the contrary, GEM delays the communication of the answers of a goal until all the possible answers have been computed, i.e., until all the branches of the partial derivation tree of the goal have been inspected. This strategy simplifies the termination detection mechanism, and we believe reduces the number of messages exchanged by principals during a computation.

In this paper we have compared GEM with other TM frameworks based on their features (Table 1). In the next future we intend to compare GEM and its variant with (some of) these frameworks in terms of performance, by means of a benchmark. With this respect, we have already implemented the basic version of the algorithm and applied it to a number of case studies in the Maritime Safety and Security (MSS) and e-Learning domain. We are currently implementing the variant of GEM with early loop detection.

9. REFERENCES

[1] M. Alves, C. V. Damásio, W. Nejdl, and D. Olmedilla. A Distributed Tabling Algorithm for Rule Based Policy Systems. In *Proc. of POLICY '06*, pages 123–132. IEEE Computer Society, 2006.

[2] K. R. Apt. Logic programming. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 493–574. MIT Press, 1990.

[3] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[4] M. Y. Becker. Cassandra: flexible trust management and its application to electronic health records. *PhD Thesis*, 2005.

[5] M. Y. Becker, J. F. Mackay, and B. Dillaway. Abductive Authorization Credential Gathering. In *Proc. of POLICY'09*, pages 1–8. IEEE Computer Society, 2009.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of S&P'96*, pages 164–173. IEEE Computer Society, 1996.

[7] W. Chen and D. S. Warren. Tabled Evaluation With Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[8] S. Costantini. Comparing different graph representations of logic programs under the Answer Set semantics. In *Proc. of ASP'01*, 2001.

[9] M. Czenko and S. Etalle. Core TuLiP Logic Programming for Trust Management. In *Proc. of ICLP'07*, LNCS 4670, pages 380–394. Springer, 2007.

[10] M. Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. H. den. Nonmonotonic Trust Management for P2P Applications. In *Proc. of STM'05*, pages 101–116. Elsevier Science, 2005.

[11] C. V. Damásio. A Distributed Tabling System. In *Proc. of TAPD'00*, pages 65–75, 2000.

[12] C. Dong and N. Dulay. Shinren: Non-monotonic trust management for distributed systems. In *Proc. of IFIPTM'10*, pages 125–140. Springer Boston, 2010.

[13] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991.

[14] R. Hu. *Efficient tabled evaluation of normal logic programs in a distributed environment*. PhD thesis, 1997.

[15] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proc. of VLDB'89*, pages 87–96. Morgan Kaufmann Publishers Inc., 1989.

[16] T. Jim and D. Suciu. Dynamically distributed query evaluation. In *Proc. of PODS'01*, pages 28–39. ACM, 2001.

[17] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proc. of PADL'03*, LNCS 2562, pages 58–73. Springer, 2003.

[18] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[19] F. Protti and G. Zaverucha. On the Relations between Acceptable Programs and Stratifiable Classes. In *Proc. of SBIA '98*, pages 141–150. Springer-Verlag, 1998.

[20] H. Przymusinska and T. C. Przymunsinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13(1):51–65, 1990.

[21] T. Przymusinski. The Well-Founded Semantics Coincides with Three-Valued Stable Semantics. *Fundamenta Informaticae*, 13:445–463, 1990.

[22] R. Rocha, F. Silva, and V. S. Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proc. of TAPD'00*, pages 77–87, 2000.

[23] K. Stine, R. Kissel, W. C. Barker, A. Lee, and J. Fahlsing. Guide for Mapping Types of Information and Information Systems to Security Categories. Special Publication SP 800-60 Rev. 1, National Institute of Standards and Technology (NIST), 2008.

[24] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated Trust Negotiation. In *Proc. of DISCEX'00*, volume 1, pages 88–102. IEEE Computer Society, 2000.

[25] M. Winslett. An Introduction to Trust Negotiation. In *Proc. of iTrust'02*, LNCS 2692, pages 275–283. Springer, 2003.

[26] C. C. Zhang and M. Winslett. Distributed Authorization by Multiparty Trust Negotiation. In *Proc. of ESORICS'08*. Springer-Verlag, 2008.

APPENDIX

A. PROOFS

DEFINITION 8 (RANKING). *Let S be the set of tables resulting from running GEM on a goal G w.r.t. $P_1, \ldots, P_n$. Let G' be a goal whose table is in S. Let θ be a solution of G' using clause $H \leftarrow B_1, \ldots, B_n$. Then, by construction $\exists \theta_0, \ldots, \theta_n$ s.t. $\theta_0 = mgu(G', H)$ and θ_j is a solution of $B_j\theta_0 \ldots \theta_{j-1}$ (with $j \in \{1, \ldots, n\}$). The ranking of θ is defined inductively as follows:*

- *$rank(\theta) = 1$ if $n = 0$ (i.e., the clause is a fact),*
- *$rank(\theta) = 1 + max(rank(\theta_1), \ldots, rank(\theta_n))$ otherwise, where $rank(\theta_j)$ is the ranking of solution θ_j.*

Proof (sketch) of Theorem 1. We proceed by contradiction and assume that there exists at least a "wrong" solution $\theta_{i,j}$ in $Sol_i \in \{Sol_1, \ldots, Sol_k\}$, i.e., a solution s.t. there is no corresponding SLD derivation of $P \cup \{G_i\}$ with c.a.s. σ s.t. $G_i\theta_{i,j}$ is a renaming of $G_i\sigma$ (*Hypothesis*).

Let us choose $\theta_{i,j}$ to be a "wrong" solution with minimal ranking (*). Since $\theta_{i,j}$ is a solution of G_i, there exists a partial tree of G_i in S created by CREATE TABLE with root $\langle id, G_i \leftarrow G_i, new\rangle$, a subnode with clause $c = H \leftarrow B_1, \ldots, B_n$ and substitutions $\theta_0, \ldots, \theta_n$ s.t. $\theta_0 = mgu(G_i, H)$, and for each $l \in \{1, \ldots, n\}$ there exists:

- A node in the partial tree of G_i with selected atom $B_l\theta_0 \ldots \theta_{l-1}$ (ACTIVATE NODE).
- A partial tree of $B_l\theta_0 \ldots \theta_{l-1}$ created by CREATE TABLE at the location of $B_l\theta_0 \ldots \theta_{l-1}$.
- A solution θ_l of $B_l\theta_0 \ldots \theta_{l-1}$; the answer $B_l\theta_0 \ldots \theta_l$ is sent to the requester of $B_l\theta_0 \ldots \theta_{l-1}$ by GENERATE RESPONSE.
- A node with clause $(H \leftarrow B_{l+1}, \ldots, B_n)\theta_0 \ldots \theta_l$ added to the partial tree of G_i by PROCESS RESPONSE.

Then, $\theta_{i,j} = \theta_0 \ldots \theta_n$. If the body of c is empty, the proof is straightforward, and omitted. Now, for each $l \in \{1, \ldots, n\}$, $rank(\theta_l) < rank(\theta_{i,j})$. So, by the minimality argument (*), for each $l \in \{1, \ldots, n\}$ there exists an SLD derivation of $P \cup \{B_l\theta_0 \ldots \theta_{l-1}\}$ with c.a.s. σ_l s.t. $B_l\sigma_0 \ldots \sigma_{l-1}\sigma_l = B_l\theta_0 \ldots \theta_{l-1}\theta_l$. But then, by standard LP results (given the presence of clause c), there exists a successful SLD derivation of $P \cup \{G_i\}$ with c.a.s. σ s.t. $G_i\sigma = G_i\theta_{i,j}$, contradicting the hypothesis. □

Proof (sketch) of Theorem 2. We proceed by contradiction, and assume that S is missing a solution of G_1. That is, there exists a successful SLD derivation of $P \cup \{G_1\}$ with c.a.s. θ and there is no solution σ of G_1 generated by the algorithm s.t. $G_1\theta = G_1\sigma$. (*Hypothesis*) This implies that there exist a (maximal) set of goals $G_1, \ldots, G_k$ in S s.t. for each $i \in \{1, \ldots, k\}$ there is a non-empty maximal set of substitutions $\{\theta_{i,1}, \ldots, \theta_{i,m_i}\}$ s.t.:

- G_i is a goal in S.
- $\theta_{i,1}, \ldots, \theta_{i,m_i}$ are correct solutions of G_i according to SLD resolution: for each $\theta_{i,j}$ there exists a successful SLD derivation of $P \cup \{G_i\}$ with c.a.s. $\theta_{i,j}$ (up to renaming).
- The algorithm does not generate the answers $G_i\theta_{i,1}, \ldots, G_i\theta_{i,m_i}$ (up to renaming).

The set $G_1, \ldots, G_k$ is not empty as it contains at least G_1.

For each i, j, let $der_{i,j}$ be the shortest SLD derivation of $P \cup \{G_i\}$ with c.a.s. $\theta_{i,j}$. Let us choose integers p, q in such a way that $der_{p,q}$ has minimal length among the derivations in the set $\{der_{i,j}\}$. The fact that $der_{p,q}$ has minimal length implies that for any goal G' in S, the following holds: if there exists an SLD derivation of $P \cup \{G'\}$ of length smaller than $len(der_{p,q})$ with c.a.s. θ', then the algorithm generates a solution ϑ' for which $G'\theta' = G'\vartheta'$ (*).

Let c be the clause used in the first step of the derivation $der_{p,q}$. If c is a fact, we immediately have a contradiction: since G_p is a goal in S, this means that there exists a partial tree of G_p created by CREATE TABLE with root node $\langle id, G_p \leftarrow G_p, new\rangle$ and a node with clause c as subnode of the root node. Therefore, the algorithm will compute a c.a.s. equivalent to $\theta_{p,q}$ (ACTIVATE NODE), contradicting the hypothesis.

If c is a rule $H \leftarrow B_1, \ldots, B_n$, and $\sigma_0 = mgu(G_p, H)$, then by hypothesis there exist SLD derivations $der_{B_1}, \ldots, der_{B_n}$, and substitutions $\sigma_1, \ldots, \sigma_n$ s.t. $H\sigma_0 \ldots \sigma_n = G_p\theta_{p,q}$, and for each $i \in \{1, \ldots, n\}$:

- der_{B_i} is an SLD derivation of $P \cup \{B_i\sigma_0 \ldots \sigma_{i-1}\}$.
- The c.a.s. of der_{B_i} is σ_i, and $len(der_{B_i}) < len(der_{p,q})$. (**)

Since G_p is a goal in S, there exists a partial tree of G_p created by CREATE TABLE with root node $\langle id, G_p \leftarrow G_p, new\rangle$ and a node with clause c as subnode of the root node. Then, it is easy to see that for each $i \in \{1, \ldots, n\}$:

- There exists a node in the partial tree of G_p with selected atom $B_i\sigma_0 \ldots \sigma_{i-1}$ (ACTIVATE NODE).
- There exists a partial tree of $B_i\sigma_0 \ldots \sigma_{i-1}$ created by CREATE TABLE at the location of $B_i\sigma_0 \ldots \sigma_{i-1}$.
- Since $len(der_{B_i}) < len(der_{p,q})$, by (*) and (**) the algorithm computes a solution equivalent to σ_i of the goal $B_i\sigma_0 \ldots \sigma_{i-1}$.
- The answer $B_i\sigma_0 \ldots \sigma_i$ is sent to the requester of $B_i\sigma_0 \ldots \sigma_{i-1}$ by GENERATE RESPONSE.
- There exists a node with clause $(H \leftarrow B_{i+1}, \ldots, B_n)\sigma_0 \ldots \sigma_i$ added to the partial tree of G_p by PROCESS RESPONSE.

Therefore, $\sigma_1, \ldots, \sigma_n$ is (equivalent to) a solution of the partial tree of G_p, contradicting the hypothesis. □

Proof (sketch) of Theorem 3. We assume that nodes (i.e., goals) in the dependency graph of P inherit the identifier (and the associated ordering) of the request for which they are created. Termination follows from two observations: (i) the dependency graph of P is finite, and (ii) the number of response messages exchanged by the principals involved in the evaluation of G is finite.

The dependency graph of P is finite (i) for the following reasons:

1. The set of goals over predicates in P (up to renaming) is finite. This is because terms that are not variables are constants in P.
2. There is no infinite path in the goal dependency graph of P composed of nodes $id_1, \ldots, id_n$ s.t. $id_n \sqsubset \ldots \sqsubset id_1$. This is because of (1) and because the algorithm never creates a new node with identifier id_i for a goal if a node with identifier id_j already exists for a variant of that goal and $id_i \sqsubset id_j$.
3. The outdegree of each node in the goal dependency graph of P is finite.

The number of response messages is finite (ii) because:

1. The (possibly empty) set of answers of a goal are transmitted only when a node for the goal is first created or new answers of its subgoals are received.
2. For any nodes id_1 and id_2, a set of answers that flows from id_2 to id_1 in response to a request id_2 never contains answers previously communicated in response to request id_2.
3. An empty set of answers may flow from id_2 to id_1 only if $id_2 \sqsubset id_1$, or id_1 identifies a lower request and a loop id_2 has just been identified.
4. There is no infinite path composed of nodes $id_n, \ldots, id_1$ in the goal dependency graph of P through which the answers flow s.t. $id_n \sqsubset \ldots \sqsubset id_1$. □

GEM + Early Loop Detection. The soundness, completeness, and termination properties of basic GEM are preserved by this variant of the algorithm. Since the way in which answers are computed has not been modified, the soundness result still holds. Completeness is guaranteed by the finalization step. In particular, the modified GENERATE RESPONSE procedure sends the answers to higher requests at every iteration of a loop, and sends the answers to side requests when the loop in which they are involved is the next to be processed. Since every loop (except for the highest) is finalized each time a fixpoint is reached, eventually every side request will receive the answers of the requested goal. In the proof of termination, we change step (3) in part (ii) as follows: an empty set of answers may flow from id_2 to id_1 only if $id_2 \sqsubset id_1$, $id_2 \hookrightarrow id_1$, or id_1 identifies a lower request and a loop id_2 has just been identified. In addition, a finalization message may flow from id_2 to id_1 only if $id_2 \sqsubset id_1$, $id_2 \hookrightarrow id_1$, or id_1 identifies a lower request and a fixpoint for loop id_2 has just been reached. Between any two finalization steps a new answer must be computed by some goal in the SCC.