TLP **14** (3): 293–337, 2014. © Cambridge University Press 2012
doi:10.1017/S1471068412000397 First published online 3 December 2012

GEM: A distributed goal evaluation algorithm for trust management

DANIEL TRIVELLATO and NICOLA ZANNONE
Eindhoven University of Technology, Eindhoven, The Netherlands
(*e-mail:* {d.trivellato,n.zannone}@tue.nl)

SANDRO ETALLE
Eindhoven University of Technology, Eindhoven, The Netherlands
and
University of Twente, Enschede, The Netherlands
(*e-mail:* s.etalle@tue.nl)

submitted 30 May 2011; revised 13 March 2012; accepted 24 September 2012

Abstract

Trust management is an approach to access control in distributed systems where access decisions are based on policy statements issued by multiple principals and stored in a distributed manner. In trust management, the policy statements of a principal can refer to other principals' statements; thus, the process of evaluating an access request (i.e., a goal) consists of finding a "chain" of policy statements that allows the access to the requested resource. Most existing goal evaluation algorithms for trust management 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 trust management systems that relies on function-free logic programming for the specification of policy statements. GEM detects termination in a completely distributed way without disclosing intensional policies, thereby preserving their confidentiality. We demonstrate that the algorithm terminates and is sound and complete with respect to the standard semantics for logic programs.

KEYWORDS: trust management, distributed goal evaluation, policy confidentiality

1 Introduction

The widespread availability of the Internet has led to a significant increase in the number of collaborations, services, and transactions carried out over networks spanning multiple administrative domains (e.g., web services). Such collaborations are frequently characterized by the interaction of users and institutions (hereafter indistinctly referred to as *principals*) who do not know each other beforehand. For this reason, in such distributed settings, attribute-based approaches to access control are mostly preferred to identity-based solutions (Ellison *et al.* 1999). Consider, for instance, an international medical research project *Alpha* involving several companies worldwide. Project *Alpha* is funded and coordinated by the multinational pharmaceutical company *mc* which, among its tasks, appoints the partners of the project consortium. In this scenario, it is likely that company *mc* does not know the project members of each partner company personally, i.e., does not know their identity. Therefore, rather than on the identity of the project members, the policies regulating the access to project's documents will be based on their attributes (e.g., project membership, specialization) and their relationships with other principals (e.g., partner companies, departments within a company).

Trust management is an approach to access control in distributed systems where access decisions are based on the attributes of principals, which are attested by digitally signed certificates called *digital credentials* (Blaze *et al.* 1996). Digital credentials (or simply credentials) are the digital counterpart of paper credentials. Credentials are defined and derived by means of policy statements that specify the conditions upon which a credential is issued, where conditions are in turn represented by credentials. A distinguishing ingredient of trust management is that all the principals in a distributed system are free to define such policy statements and determine where to store them. The set of policy statements defined by a principal forms the *policy* of that principal. In the scenario above, for instance, the rules of company *mc* dictating the conditions for the membership of a user to project *Alpha* (e.g., a Master degree in chemistry) form the policy of *mc*. These statements can be stored by *mc* or at another principal's location (e.g., by each partner company).

In trust management languages, policy statements are often expressed as Horn clauses (Li and Mitchell 2003) where each atom represents a credential, and is possibly annotated with the storage location of the statements defining the credential. Depending on the language, the location can be expressed implicitly (Li *et al.* 2003; Czenko and Etalle 2007) or explicitly (Becker 2005; Alves *et al.* 2006). While typically principals do not have direct access to each other' policies, the statements of a principal can refer to other principals' policies, thereby delegating authority to them. For instance, assume that a hospital *h* authorizes the members of project *Alpha* certified by the local pharmaceutical company *c*1 to access the (anonymized) medical records of its patients suffering from genetic diseases. The policies governing this scenario can be represented by the following clauses:

1. mayAccessMedRec(h,X) ← memberOfAlpha($c1$,X).
2. memberOfAlpha($c1$,X) ← projectPartner(mc,Y), memberOfAlpha(Y,X).
3. projectPartner(mc,$c2$).
4. projectPartner(mc,$c3$).
5. projectPartner(mc,$c4$).
6. memberOfAlpha($c2$,X) ← memberOfAlpha($c1$,X).
7. memberOfAlpha($c2$,$alice$).
8. memberOfAlpha($c3$,bob).
9. memberOfAlpha($c4$,$charlie$).

where the first parameter of each atom denotes the location where the credential that the atom represents is defined. Here, hospital *h* relies on the policy statements of *c*1 to determine who is authorized to access the hospital's medical records (clause 1); in turn, $c1$ relies on the policy statements of *mc* and the partner companies appointed by *mc* for the definition of project *Alpha*'s members (clause 2). Therefore, the process of evaluating a request to access the hospital's records (i.e., a *goal*) consists of deriving a "chain" of policy statements delegating the authority from hospital *h* (i.e., the resource owner) to the members of project *Alpha* (i.e., the authorized principals). This process, referred to as *credential chain discovery* (Li *et al.* 2003), can be addressed using *goal evaluation algorithms.*

Since in trust management policies are stored at different locations, goal evaluation algorithms require principals to disclose their policy statements to other principals to enable credential chain discovery. In particular, for a successful computation, principals need to disclose at least (part of) their *extensional policy*, that is, the credentials that can be derived from their policy and are required for an access decision. For example, suppose that hospital *h* wants to determine who can access the medical records of its patients. To compute the answers of this goal, it is clear that $c1$ has to disclose to *h* the credentials certifying all the project members. Most of the existing goal evaluation algorithms (e.g., Li *et al.* 2003; Czenko and Etalle 2007), however, rely on a centralized evaluation strategy and require principals to disclose also (part of) their *intensional policy*, i.e., the policy statement used to derive those credentials (e.g., clause 2 in the example policy).

We argue that one of the advanced desiderata of goal evaluation algorithms for trust management is that the amount of information about intensional policies that principals reveal to each other should be minimized. In fact, intensional policies might contain sensitive information about the relationships among principals, whose disclosure would leak valuable business information that can be exploited by other principals in the domain (e.g., rival companies) (Yu and Winslett 2003). For example, if $c1$'s policy was disclosed to other principals for evaluation (e.g., to hospital *h*), the involvement of *mc* in project *Alpha* along with the list of all project partners would be exposed. As a consequence, some competitors of *mc* could start investing on similar projects, or could try to get at the project members to acquire sensitive information and project results. Furthermore, the loss of confidentiality of intensional policies can result in attempts by other principals to influence the policy evaluation process (Stine *et al.* 2008), and allows adversaries to know what credentials they need to forge to illegitimately get access to a resource (Frikken *et al.* 2006).

To protect the confidentiality of intensional policies, it is necessary to design a completely distributed goal evaluation algorithm that discloses as few information on intensional policies as possible. Since bottom-up approaches to goal evaluation [e.g., fixpoint semantics (Park 1969), magic templates (Ramakrishnan 1991) and magic sets (Chen 1997)] require knowledge of all the policy statements that depend on a given credential, they do not represent an applicable solution to our problem. Hence, a top-down approach to goal evaluation needs to be employed. The design of a distributed top-down algorithm, however, requires addressing two main problems: (a) loop detection and (b) termination detection. In addition, to reduce network overhead, the goal evaluation algorithm should attempt to decrease the number of messages that principals exchange.

$$\begin{tikzcd}
mayAccessMedRec(h1,X) \arrow[r] & memberOfAlpha(c1,X) \arrow[r] \arrow[dl] \arrow[d] \arrow[dr] & projectPartner(mc,Y) \\
memberOfAlpha(c2,X) \arrow[ur, bend left] & memberOfAlpha(c3,X) & memberOfAlpha(c4,X)
\end{tikzcd}$$

Fig. 1. Call graph of the evaluation of the example policies.

Loops are formed when the evaluation of a goal leads to a new request for the same goal. In our scenario, for example, to determine the set of project members without disclosing its intensional policy to hospital h, $c1$ should first request the list of project partners to mc, and then the list of their project members to $c2$, $c3$, and $c4$. Since $c2$ in turn relies on the policy statements of $c1$ to determine its project members, $c2$ would pose the same request back to $c1$, forming a loop. Figure 1 shows the "call graph" originating from this sequence of requests. Intuitively, $c1$ should detect the loop and refrain from evaluating $c2$'s request, as doing so could lead to a non-terminating chain of requests. Even though in the example scenario loops could be avoided, for instance, by requiring a single company (e.g., mc) to define the set of project members, this cannot be guaranteed in distributed systems characterized by the absence of a coordinating principal. Examples of such scenarios include self-organizing networks (Di Marzo Serugendo *et al.* 2004) and access control policies based on independent information sources [e.g., the Friend of a Friend – FOAF – project (http://www.foaf-project.org/)].

Existing goal evaluation algorithms employ *tabling* techniques (Tamaki and Sato 1986; Vieille 1987; Bry 1990; Chen and Warren 1996) for the detection of loops. Although some of these algorithms resort to a distributed tabling strategy (e.g., Hu 1997; Damásio 2000), they rely on centralized data structures to detect termination – i.e., to detect when all the answers have been collected – thus leaking some policy information. In fact, the real challenge in designing a goal evaluation algorithm that does not disclose intensional policies lies in detecting termination distributedly. In the example, we have the following possible answer flow: $c3$ returns *bob* as answer to $c1$, which forwards it to h and $c2$; $c4$ returns *charlie* as answer to $c1$; $c1$ sends *charlie* as additional answer to h and $c2$; $c2$ returns *alice*, *bob*, and *charlie* as answer to $c1$, which sends *alice* as additional answer to h and $c2$. At this point, all the requests have been fully evaluated, 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 the context of access control, where a decision (positive or negative) always needs to be made. A few top-down goal evaluation algorithms are able to detect the termination of a computation distributedly (Alves *et al.* 2006; Zhang and Winslett 2008); however, they do not detect when the single goals within a computation are fully evaluated. In top-down goal evaluation, detecting when the evaluation of a goal has terminated is necessary to allow (a) for memory deallocation and (b) the use of negation, which is employed by some systems to express non-monotonic constraints (e.g., separation of duty) (Czenko *et al.* 2006; Dong and Dulay 2010).

Finally, 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 the principal has computed it, as done, for example, in Alves *et al.* (2006). This is, however, suboptimal from the viewpoint of network overhead; in the example above, $c1$ eventually sends three distinct messages to h and $c2$, one for each answer. A more network-efficient 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. In a trust management system, where network latency is likely to be a bigger bottleneck than computational power, it is preferable to have a mechanism that allows principals to wait until they collect the maximum possible set of answers before sending them to the requester, while avoiding deadlocks. Even though this solution may delay the identification of the answers of ground goals (i.e., goals expecting a single answer), the "superfluous" computed answers might become relevant for the evaluation of other goals, reducing the delay in future computations.

In this paper we present GEM, a goal evaluation algorithm for trust management systems that addresses all the above-mentioned problems. In GEM, policy statements are expressed as function-free logic programming clauses; each statement is stored by the principal defining it. GEM computes the answers of a goal in a completely distributed way without disclosing intensional policies of principals, thereby preserving their confidentiality. The algorithm deals with loops in three steps: (1) detection, (2) processing, and (3) termination. To enable loop detection, we employ a distributed tabling strategy and associate an identifier to each request for the evaluation of a goal. After its detection, a loop is processed by iteratively evaluating the goals in the loop until a fixpoint is reached, i.e., no more answers of the goals in the loop are computed, at which point their evaluation is terminated. This three-steps approach enables GEM to detect both when the whole computation has terminated, and when the single goals within a computation are fully evaluated, allowing for the use of non-monotonic constraints in policies. In addition, by exploiting the information stored in the table of a goal, principals are able to delay the response to a request until a "maximal" set of answers of the goal has been computed without running the risk of deadlocks. We demonstrate that GEM terminates and is sound and complete with respect to the standard semantics for logic programs.

The remainder of the paper is structured as follows. Section 2 presents preliminaries on logic programming and Selective Linear Definite (SLD) resolution. Section 3 introduces GEM and its implementation. Section 4 demonstrates the soundness, completeness, and termination of the algorithm, and discusses what information is disclosed by GEM during the evaluation of a goal. Section 5 presents the results of experiments conducted to evaluate the performance of GEM. A possible extension of GEM to deal with negation is presented in Section 6. Section 7 discusses related work. Finally, Section 8 concludes and gives directions for future work.

2 Preliminaries on logic programming

In this section we revisit the concepts of logic programming (Apt 1990) that are relevant to this paper. In particular, we review function-free logic programs.

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 or constants). An atom is *ground* if $t_1, \ldots, t_n$ are constants. A *clause* is an expression of the form $H \leftarrow B_1, \ldots, B_n$ (with $n \geqslant 0$), 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 A is *defined in the program* P if and only if there is a clause in P that has an atom A' in its head such that A and A' are unifiable. Finally, a *goal* is a clause with no head atom, i.e., a clause of the form $\leftarrow B_1, \ldots, B_n$. Without loss of generality, in this paper, we restrict to goals with $0 \leqslant n \leqslant 1$, that is, consisting of at most one atom. The empty goal is denoted by $\emptyset$.

SLD resolution (Selective Linear Definite clause resolution) (Kowalski 1974) is the standard operational semantics for logic programs. In this paper, we refer to 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 $G_0 = \leftarrow B_1, \ldots, B_n$ and a clause c in a program P. Let $H \leftarrow B'_1, \ldots, B'_m$ be a variant of c variable disjoint from $\leftarrow B_1, \ldots, B_n$. Let B_1 and H unify with *most general unifier* (*mgu*) θ. The goal $G_1 = \leftarrow (B'_1, \ldots, B'_m, B_2, \ldots, B_n)\theta$ is called a *resolvent of* G_0 *and* c *with selected atom* B_1 *and mgu* θ. An SLD *derivation step* is denoted by $G_0 \xrightarrow{\theta} G_1$. Clause $H \leftarrow B'_1, \ldots, B'_m$ is called *input clause*, and atom B_1 is called the *selected atom* of G_0.

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 $\delta := G_0 \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n} G_n$ is a finite prefix of a derivation, we say that $\theta = \theta_1 \cdots \theta_n$ is a *partial derivation* and θ is a *partial computed answer substitution* of $P \cup \{G_0\}$. If δ ends with the empty goal $\emptyset$, θ is called *computed answer substitution* (*c.a.s.*). Let $G_0 = \leftarrow B_1$. Then, we also call θ a *solution* of G_0 and $B_1\theta$ 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 technique to prevent infinite derivations is *tabling* (Tamaki and Sato 1986; Vieille 1987; Bry 1990; Chen and Warren 1996; Guo and Gupta 2001; Shen *et al.* 2001; Zhou and Sato 2003). Given a goal G_0 consisting of an atom defined in a program P, tabling-based goal evaluation algorithms create a table for each (sub)goal in the SLD resolution of $P \cup \{G_0\}$, to keep track of the previously evaluated goals and thus avoid the reevaluation of a subgoal. Tabling algorithms differ mainly in the data structures employed for the evaluation of goal G_0. Linear tabling (Shen *et al.* 2001; Zhou and Sato 2003) and Dynamic Reordering of Alternatives (DRA) (Guo and Gupta 2001), for instance, evaluate G_0 by building a single derivation tree of G_0. In SLG resolution (Chen and Warren 1996), on the other hand, goal G_0 is evaluated by producing a forest of (partial) derivation trees, one for each subgoal in the resolution of $P \cup \{G_0\}$. In SLG, 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. The evaluation of a goal by means of a forest of derivation trees proposed by SLG resolution is at the basis of the distributed goal evaluation algorithm proposed in this paper.

3 The GEM algorithm

In this section we first introduce some definitions and basic assumptions underlying our work. Then, we present GEM and discuss its implementation.

3.1 Definitions and assumptions

Similar to other works on trust management (e.g., Li *et al.* 2003; Alves *et al.* 2006), we consider policy statements expressed as function-free logic programming clauses. As in most trust management systems, policy statements are stored at different locations: each location is controlled by a principal who is responsible for defining and evaluating the policy statements at that location. We assume a one-to-one correspondence between locations and principals; accordingly, we use a principal's identifier to refer to the location she controls. To represent the location where a policy statement is stored, we require every atom to have the form $p(loc, t_1, \ldots, t_n)$, where loc is a mandatory term that represents the location where the atom is defined, and $t_1, \ldots, t_n$ are terms. For instance, $p(bob, \ldots)$ refers to p as defined by Bob and thus stored at Bob's location.

Let a be a principal in the trust management system. We call the set of policy statements defined by a the *local trust management policy* (or simply the *policy*) of principal a. The set of clauses with non-empty body in a's policy is the *intensional policy* of a, while the set of facts that can be derived from principal a's policy forms the *extensional policy* of a. The set of all the local policies in the trust management system is called *global policy*.

Since we consider the confidentiality of intensional policies to be a main concern in trust management systems, we assume that principals do not have access to the policies at other principals' locations. As a consequence, the answers of a goal cannot be derived by building the derivation tree of the goal as done by SLD resolution, as this might involve input clauses defined by different principals. Similar to SLG resolution, in GEM a principal computes the answers of a goal defined in her policy by building the *partial derivation tree* of the goal. Different from a derivation tree, in the partial derivation tree of a goal G only the first derivation step is obtained by resolution with the clauses defining G; all the subsequent steps are by substitution with the solutions of the subgoals of G.

Definition 1
Let $G = \leftarrow A$ be a goal and P_A be the policy in which A is defined. A *partial derivation tree* of G is a derivation tree with the following properties:

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

Note that, to enable the evaluation of an atom B_1 in the partial derivation tree of goal G, the location where B_1 is defined must be known by the principal evaluating G. A straightforward solution for guaranteeing that this requirement is satisfied would be to impose the location parameter of each atom in a policy to be ground at policy definition time. This, however, would limit the constraints that a principal can express. Consider, for instance, clause 2 on page 2. In the clause, the location parameter of atom *memberOfAlpha(Y,X)* is determined at runtime based on the answers of *projectPartner(mc,Y)*. Therefore, rather than relying on a "static" safety condition, we require 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*. 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* (Apt and Marchiori 1994) which guarantee that certain parameters of an atom are ground when the atom is selected for evaluation.

Finally, we define a classification criteria for goal evaluation algorithms based on disclosed policy information. We will use such criteria to compare GEM with the existing algorithms (see Section 7). The classification criteria consists of two elements: an extensional and an intensional policy confidentiality level. Intuitively, the first characterizes algorithms in terms of how much information about extensional policies they disclose during goal evaluation, while the second refers to the disclosure of intensional policies. Confidentiality levels define an increasing scale used to characterize from the most conservative approaches where no policy information is disclosed, to the least confidentiality-preserving solutions which disclose respectively extensional and intensional policies in full. The extensional and intensional confidentiality levels are presented in Figure 2. In a goal evaluation algorithm classified as E1-I2, for example, principals disclose to a requester all the

Level	Disclosed Information
E0	None
E1	Answers of a goal

(a)

Level	Disclosed Information
I0	None
I1	Part of the dependency graph
I2	Full dependency graph
I3	Clauses

(b)

Fig. 2. Classification of goal evaluation algorithms in terms of disclosed policy information. (a) Extensional policy confidentiality levels; (b) intensional policy confidentiality levels.

answers of the requested goals. In addition, all the dependencies among the goals in the global policy are disclosed to the principals responsible for goal evaluation.

3.2 Intuition

In this section we describe how GEM computes the answers of a goal. Given a goal G, GEM computes the answers of G 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 answers from each branch of the tree of G have been computed, they are sent to the principal(s) that requested the evaluation of G. G is *completely evaluated* when no more answers of G can be computed.

To illustrate how GEM works in detail, we consider the scenario presented in Section 1, where several pharmaceutical companies collaborate in the research project *Alpha*. However, we slightly modify the global policy to better focus on the algorithm's features. In particular, we assume that company $c1$ already knows which are the partner companies in project *Alpha*, without needing to request them to *mc*, and we reduce the partner companies to $c2$ and $c3$ only. Furthermore, we consider a research institute *ri* that works on project *Alpha* in partnership with company $c2$. As a result, we have the following global policy:

1. memberOfAlpha($c1$,X) $\leftarrow$ memberOfAlpha($c2$,X).
2. memberOfAlpha($c1$,X) $\leftarrow$ memberOfAlpha($c3$,X).
3. memberOfAlpha($c2$,X) $\leftarrow$ memberOfAlpha(ri,X).
4. memberOfAlpha($c2$,*alice*).
5. memberOfAlpha($c3$,*bob*).

Recall that the first parameter of the atom in the head indicates the principal storing and evaluating a clause: clauses 1 and 2 are evaluated by $c1$, clauses 3 and 4 by $c2$, and clause 5 by $c3$. Suppose that hospital h sends to company $c1$ a request for (the evaluation of) goal $\leftarrow$ *memberOfAlpha(*$c1$,X*)* (for a matter of readability, from here on we omit the $\leftarrow$ symbol when referring to a goal $\leftarrow A$, and we simply refer to it as A). Figure 3 shows the *call graph* of the evaluation of *memberOfAlpha(*$c1$,X*)* with respect to the example global policy. A call graph is a directed graph where nodes represent goals and edges connect each goal to its subgoals (Leuschel *et al.* 1998). In other words, edges represent (evaluation) requests.

GEM performs a depth-first computation. When $c1$ receives the initial goal, it evaluates the first applicable clause in its policy (i.e., clause 1) and sends a request

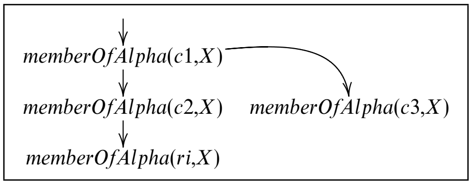

Fig. 3. Call graph of the example global policy.

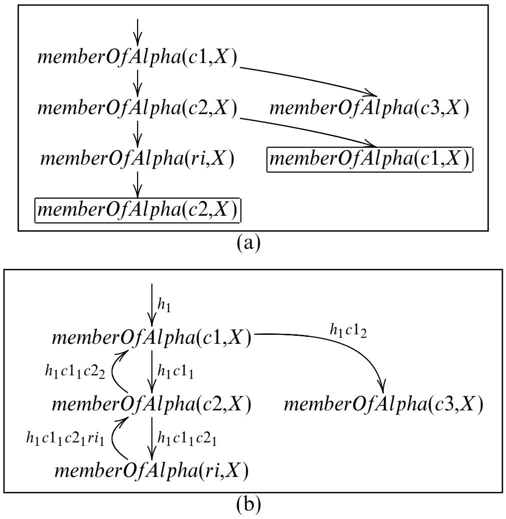

Fig. 4. Call graph of the example global policy with loops. (a) Call graph with loops; (b) compact call graph with request identifiers and loops.

for *memberOf-Alpha(c2,X)* to *c*2. In turn, *c*2 sends a request for *memberOfAlpha(ri,X)* to *ri*. *ri* does not have any clause applicable to *memberOfAlpha(ri,X)* and returns an empty set of answers to *c*2. *c*2 evaluates the next applicable clause (i.e., *memberOfAlpha(c2,alice)*), which is a fact. Since *c*2 does not have any other clause to evaluate, it sends the computed answer to *c*1. *c*1 applies the next clause (clause 2) and sends a request for *memberOfAlpha(c3,X)* to *c*3, that returns answer *memberOfAlpha(c3,bob)* to *c*1 after applying clause 5. At this point, *memberOfAlpha(c1,X)* is completely evaluated and *c*1 sends answers *memberOfAlpha(c1, alice)* and *memberOfAlpha(c1,bob)* to hospital *h*.

The evaluation of a subgoal of a goal *G*, however, may lead to new requests for *G*, forming a *loop*. In our scenario, this reflects the "sharing" of project members among partner companies. Consider, for instance, the global policy above with the following two additional clauses, stored by *c*2 and *ri* respectively:

6. memberOfAlpha($c2,X$) $\leftarrow$ memberOfAlpha($c1,X$).
7. memberOfAlpha(ri,X) $\leftarrow$ memberOfAlpha($c2,X$).

The new call graph is shown in Figure 4(a). Now, when *ri* receives the request for *memberOfAlpha(ri,X)* from *c*2, it applies clause 7 and sends a request for *memberOfAlpha(c2,X)* back to *c*2, forming a loop. Similarly, the evaluation of clause 6 by $c2$ leads to another loop. The requests forming a loop are identified by boxed atoms in Figure 4(a). Formally, a loop is defined as follows.

Definition 2
Let C be the call graph of the evaluation of a goal G with respect to a global policy P. A *loop* is a maximal subgraph of C consisting of goals $G_1, \ldots, G_k$ such that for each $G_i \in \{G_1, \ldots, G_k\}$ there exists a path that leads from G_1 to G_i and from G_i to (a variant of) G_1. Then, we say that goals $G_1, \ldots, G_k$ are *involved* in the loop. □

Intuitively, requests forming a loop should not be further evaluated. However, in the example above, $c1$ and $c2$ 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 algorithms (e.g., Chen and Warren 1996; Damásio 2000; Li *et al.* 2003), loop detection (and termination) is made possible by the system's "global view" on the derivation process. For example, centralized goal evaluation algorithms such as SLG (Chen and Warren 1996) and RT (Li *et al.* 2003) identify loops by observing goal dependencies respectively in the call stack and in the call graph of the global policy. In a similar way, the distributed algorithm proposed by Damásio (2000) requires the dependency graph of the global policy to be known to all principals. Such global view, however, implies the loss of policy confidentiality. GEM detects loops and their termination in a completely distributed way without resorting to any centralized data structure. In GEM, 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). An SCC is a set of mutually dependent subgoals. More precisely, a set of goals $G_1, \ldots, G_k$ is part of a SCC if for each $G_i \in \{G_1, \ldots, G_k\}$ there exists a goal $G_j \in \{G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_k\}$ such that G_i and G_j are involved in a common loop. To enable the identification of SCCs, we assign to each request a unique identifier from an *identifier domain*.

Definition 3
An *identifier domain* is a triple $\langle I, \sqsubset, \hookrightarrow \rangle$, where:

- I is a set of sequences 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.
- The following property holds: $\forall id_1, id_2, id_3, id_4 \in I$ if $id_1 \sqsubset id_2$, $id_3 \sqsubset id_4$, and $id_2 \hookrightarrow id_4$, then $id_1 \hookrightarrow id_3$. □

Intuitively, $\sqsubset$ defines a top-down ordering and $\hookrightarrow$ defines a left-to-right ordering with respect to the call graph of the global policy. In other words, $\sqsubset$ reflects the order in which the subgoals in a branch of the graph are evaluated, whereas $\hookrightarrow$ reflects the order in which the branches are inspected. Several identifier domains can be employed whose identifiers respect these partial orders (e.g., based on alphanumeric ordering). For the sake of simplicity, in the following 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 of G has the form id_0s_1, denoting the concatenation of id_0 with a sequence of characters s_1. Then, $\sqsubset$ is a prefix relation, and we have that $id_0s_1 \sqsubset id_0$. Ordering $\hookrightarrow$ is a partial order on the strings composing the identifiers. For example, consider another subgoal G_2 of G with identifier id_0s_2, which is evaluated after G_1. Then, we have that $id_0s_1 \hookrightarrow id_0s_2$. Even though identifiers from this domain leak some policy information (see Section 4.2 for more details), they allow for an easy visualization of the relationships among identifiers. When applying GEM in practice, more confidentiality-preserving identifier domains can be employed.

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

Figure 4(b) represents a compact version of the call graph in Figure 4(a), where loop coordinators are depicted only once. In addition, in Figure 4(b) edges are labeled with the corresponding request identifier. In the remainder of the paper, we concatenate the identifier of a request for a goal evaluated by company $c1$ with meta-variables of the form $c1_i$. Thus, for instance, $c1_1$ and $c1_2$ are two distinct sequences of characters generated by $c1$. In the figure, identifiers h_1, h_1c1_1, h_1c1_2, and $h_1c1_1c2_1$ identify higher requests for goals *memberOfAlpha(c1,X)*, *memberOfAlpha(c2,X)*, *memberOfAlpha(c3,X)*, and *memberOfAlpha(ri,X)* respectively; identifiers $h_1c1_1c2_1ri_1$ and $h_1c1_1c2_2$ identify lower requests for goals *memberOfAlpha(c2,X)* and *memberOfAlpha(c1,X)* respectively. Goals inherit the ordering associated with the identifier of their higher request. Therefore, in Figure 4(b) goal *memberOfAlpha(c1,X)* is higher than *memberOfAlpha(c2,X)*, *memberOf-Alpha(c3,X)*, and *memberOfAlpha(ri,X)*. Goals *memberOfAlpha(c2,X)* and *memberOfAl-pha(c1,X)* are the coordinators of loops h_1c1_1 and h_1 respectively. Loop h_1c1_1 is lower than loop h_1, which is the highest loop of the SCC; therefore, *memberOfAlpha(c1,X)* is the leader of the SCC. The identifier of the lower requests enables $c1$ and $c2$ to determine the subgoals involved in the loop, which are *memberOfAlpha(c2,X)* and *memberOfAlpha(ri,X)* respectively.

Loop Processing. When a loop is detected, GEM sends the answers of the coordinator already computed 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 (a) processes the received answers, (b) "freezes" the evaluation of the subgoal involved in the loop and evaluates other branches of the partial derivation tree of the locally defined goal. Then, when all branches have been evaluated, (c) the new answers are sent to the requester of the higher request with a notification about the loop. We call the execution of actions (a), (b), and (c) a *loop iteration step*.

Definition 4
Let G be a goal, and $G_1, \ldots, G_k$ be the subgoals of G s.t. $G, G_1, \ldots, G_k$ are involved in a loop *id*. A *loop iteration step* for goal G is a three-phases process in which the principal evaluating G:

1. Receives a set of answers of the subgoals $G_1, \ldots, G_k$ of G.
2. Evaluates all the nodes in the partial derivation tree of G whose selected atom is not involved in a loop.
3. Sends the newly derived answers of G to the requester of G. □

The definition above applies to all the goals involved in a loop but the loop coordinator. The loop iteration step for a loop coordinator differs in the order in which the phases occur. In particular, for the coordinator phase (c) precedes (a) and (b), and the latter two are executed only after a loop iteration step for the other goals in the loop has been performed. In other words, the processing of the coordinator occurs only after all the other goals in the loop have been processed. This is because the coordinator, being the “highest goal” in the loop, is assigned the task of overseeing its processing. More precisely, it is in charge of starting [phase (c)] a new *loop iteration* whenever the answers of its subgoals lead to new answers of the coordinator [computed in phases (a) and (b)], i.e., until a fixpoint is reached. This difference is reflected in the definition below.

Definition 5
Let $G_1, \ldots, G_k$ be the goals involved in a loop id_1 s.t. goal G_1 is the loop coordinator. A *loop iteration* is a process where:

1. The answers of G_1 that have not been previously sent are sent to the requesters of the lower requests for G_1 (phase (c) of the loop iteration step for the coordinator).
2. For each $G_i \in \{G_2, \ldots, G_k\}$ a loop iteration step for G_i is performed, s.t. for each $G_j \in \{G_2, \ldots, G_k\}$, if G_j is lower than G_i then the loop iteration step for G_j is executed before the loop iteration step for G_i.
3. The principal evaluating G_1 receives a set of answers of the subgoals of G_1 involved in loop id_1 (phase (a) of the loop iteration step for G_1). All the nodes in the partial derivation tree of G_1 whose selected atom is not involved in a loop are processed (phase (b) of the loop iteration step). □

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. Note 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 the example [Fig. 4(b)], when $c2$ identifies loop h_1c1_1, it informs ri that they are both involved in loop h_1c1_1. Since ri has no more clauses to evaluate, it returns an empty set of answers to $c2$ notifying it that *memberOfAlpha(ri,X)* is in loop h_1c1_1. The further evaluation of *memberOfAlpha(c2,X)* leads to the identification of loop h_1 and to a new answer *memberOfAlpha(*$c2$*,alice)*, which is sent first to *ri* in the context of loop h_1c1_1. In turn, *ri* computes answer *memberOfAlpha(ri,alice)* and sends it to $c2$. Now, a fixpoint for loop h_1c1_1 has been reached and $c2$ sends *memberOfAlpha(*$c2$*,alice)* to $c1$ notifying it that *memberOfAlpha(*$c2,X$*)* is in loop h_1. Note that *memberOfAlpha(*$c2,X$*)* is also in loop h_1c1_1, but since this loop does not involve $c1$, $c1$ is not notified of it. Next, $c1$ computes answers *memberOfAlpha(*$c1$*,alice)* and *memberOfAlpha(*$c1$*,bob)* (the latter being found through the evaluation of *memberOfAlpha(*$c3,X$*)*), and sends them to $c2$ in the context of loop h_1. In turn, $c2$ computes *memberOfAlpha(*$c2$*,bob)*. Now, $c2$ has to find a fixpoint for loop h_1c1_1 given the new answer before proceeding with the evaluation of loop h_1. It is worth noting that *ri* is not aware of loop h_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 the goals in an SCC is initiated by the principal handling the leader of the SCC when a fixpoint for the loop it coordinates has been reached. In the example, when the answers of *memberOfAlpha(*$c2,X$*)* do not lead to new answers of *memberOfAlpha(*$c1,X$*)*, $c1$ informs $c2$ (which in turn informs *ri*) that the evaluation of *memberOfAlpha(*$c1,X$*)* is terminated and sends answers *memberOfAlpha(*$c1$*,alice)* and *memberOfAlpha(*$c1$*,bob)* to h.

Side Requests. So far we have only considered "linear" loops, i.e., loops formed by lower requests. However, higher requests can also lead to a loop. Consider, for instance, the following additional clause stored by company $c3$:

8. memberOfAlpha($c3,X$) $\leftarrow$ memberOfAlpha(ri,X).

The new (compact) call graph is shown in Figure 5(a). Now, the evaluation of goal *member-OfAlpha(*$c3,X$*)* by $c3$ leads to a request for goal *memberOfAlpha(ri,X)*, which is involved in loop h_1c1_1. *ri* can identify that the request originates from the evaluation of a goal in the same SCC as *memberOfAlpha(ri,X)*, since the request identifier $h_1c1_2c3_1$ is side of the identifier $h_1c1_1c2_1$ of the initial request for *memberOfAlpha(ri,X)* (i.e., $h_1c1_1c2_1 \hookrightarrow h_1c1_2c3_1$). However, it cannot identify the loop in which the goal evaluated by $c3$ is involved. This is because loop notifications are only transmitted to higher goals, and thus *ri* is not aware of loop h_1. We refer to the request from $c3$ as *side request*, and we call *memberOfAlpha(*$c3,X$*)* a *side goal.*

The main problem with side requests is that it is difficult to determine when they should be responded to. For example, if *ri* sends answers to $c3$ at every iteration of loop h_1c1_1, $c1$ would not know when to stop waiting for answers from $c3$ (since $c1$ does not know on which goals *memberOfAlpha(*$c3,X$*)* depends). On the other hand, *ri* cannot wait until a fixpoint is reached for loop h_1c1_1, since only $c2$ (the principal handling the coordinator) is aware of that. To enable the detection of termination, however, a side request should be responded to only when a fixpoint is computed for all the loops lower than the loop in which the side goal is involved.

A simple yet effective solution to this problem is to treat a side request for a goal as a "new" request (i.e., a request for a goal that has not yet been evaluated) and

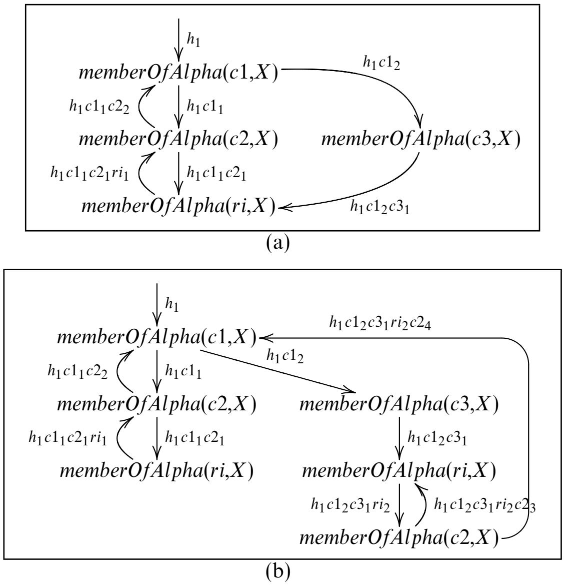

Fig. 5. Call graphs for side requests. (a) Call graph with side request; (b) unfolded call graph.

to reevaluate the goal. Accordingly, when a side request is received, GEM creates a new partial derivation tree for the goal and proceeds with its evaluation. This corresponds to inspecting multiple times some branches of the call graph of the program [Fig. 5(b)]; however, it allows us to obtain a call graph formed only by linear loops which, as shown previously in this section, can be successfully evaluated by GEM. Note that, despite in the unfolded graph in Figure 5(b) some nodes and edges are duplicated with respect to the folded graph in Figure 5(a), the flow of answers among goals is equivalent. This can be easily seen by the fact that edges connect the same nodes in both call graphs. In Section 4 (Theorem 3) we show that a call graph is always finite.

Even though possible in theory, we expect side requests not to occur frequently in the evaluation of a policy. For instance, no side request was present in any of the example policies in the literature. Therefore, we believe the overhead imposed by the proposed solution to be negligible in practice. Nevertheless, we point out that the size of the unfolded graph is in the worst case exponential with respect to the number of nodes in the original graph. The reevaluation of side requests, in fact, resembles the approach adopted by SLD resolution, which reevaluates every goal encountered during a computation. However, thanks to our ability to detect loops, the number of goals evaluated by GEM during a computation is never higher than the number of goals that would be evaluated using SLD resolution.

Since we treat side requests in the same way as new requests, the partial order $\hookrightarrow$ is not exploited by the version of GEM presented here. An alternative solution that prevents the reevaluation of side requests and thus requires their identification could be achieved by transmitting loop notifications to the requesters of both higher and lower requests, so that all the principals evaluating a goal in an SCC would be aware of the identifiers of all the loops in the SCC. Furthermore, when a fixpoint for a loop is reached, the principal handling the loop coordinator should start a new loop iteration to inform all the goals in the loop that the fixpoint has been reached. These two modifications would enable principals that receive a side request to know (a) in which loop the side request is involved and (b) when to reply to the side request, that is, when all the loops lower than the one in which the side goal is involved have reached a fixpoint. Given the complexity of the solution, in this paper we present only the "basic" version of GEM, which reevaluates side requests; the implementation of the described optimization is subject of future work.

To conclude, we point out that evaluating every higher request for a goal that is not yet completely evaluated is fundamental for enabling loop detection. Consider, for instance, a global policy consisting only of clauses 1 and 6 on pages 9 and 10 respectively. Assume that hospital *h* issues at the same time a request for goals *memberOfAlpha(c*1*,X)* and *memberOfAlpha(c*2*,X)* with identifier id_1 and id_2 respectively. The evaluation of the request by principals *c1* and *c2* leads to a higher request for goals *memberOfAlpha(c*2*,X)* and *memberOfAlpha(c*1*,X)* respectively. When these second higher requests are received, a partial derivation tree for the requested goals already exists. If the requests were not further processed, the loop identification would not be possible as no lower request would be issued, and the computation would deadlock. Therefore, both initial requests must be processed independently. This "problem" is common to all the distributed goal evaluation algorithms whose termination detection exploits request identifiers (e.g., Alves *et al.* 2006). Even though relatively simple solutions to this problem can be found (e.g., using timestamped requests, where only the oldest is evaluated), in this paper we focus on a "basic" solution for distributed goal evaluation, and do not address efficiency-related issues.

3.3 Implementation

Here, we introduce the data structures and procedures used by GEM to evaluate a goal.

Data Structures. In GEM, principals communicate by exchanging *request* and *response* messages. We rely on blocking communication, that is, whenever a principal *a* sends (respectively receives) a request or response message, no other operation is performed by *a* until the sending (resp. receipt) process is completed. In addition, we assume that a message sent by principal *a* to a principal *b* is always received (once) by principal *b*.

Definition 6
A *request* is a triple $\langle id, req, G\rangle$, where:

- *id* is the request identifier;
- *req* is the principal issuing the request, called *requester*;
- *G* is a goal $\leftarrow p(loc, t_1, \ldots, t_n)$, 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 an identifier *id* and is sent to the principal defining *G*.

Definition 7
Let $r = \langle id, req, G\rangle$ 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;
- $S_{ans} \in \{active, loop(id_1), disposed\}$ is the status of the evaluation of G, where id_1 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 G (Ans) together with the status of G's evaluation (S_{ans}) and information about the loops in which it is involved ($Loops$). S_{ans} is *disposed* if G has been completely evaluated, *active* if additional answers of G may be computed, and $loop(id_1)$ if the response is sent in the context of the evaluation of loop id_1.

As discussed in Section 3.2, GEM computes the answers of a goal by a depth-first evaluation of its partial derivation tree (Definition 1), which may involve the generation of requests for subgoals evaluated at different locations. In the implementation, we represent a partial derivation tree as a data structure called *evaluation tree*. Compared to partial derivation trees, an evaluation tree keeps track of the identifier of the request and status of the evaluation of the selected atom of each *node* in the evaluation tree.

Definition 8
A *node* is a triple $\langle id, c, S\rangle$, where:

- id is the node identifier;
- c is a 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. It is set to *active* when a body atom is selected, and to *disposed* when the selected atom is completely evaluated. The status is set to *loop(ID)* if the selected atom is involved in some loops, where ID is the set of identifiers of those loops, and to *answer* if c is a fact. As mentioned in Section 2, we employ the leftmost selection rule. Thus, the selected atom of c is always the first body atom.

Definition 9
The *evaluation tree* of a goal $G = \leftarrow A$ is a tree with the following properties:

- the root is node $\langle id_0, A \leftarrow A, S_0\rangle$;
- there is an edge from the root to a node $\langle id_1, (A \leftarrow B_1, \ldots, B_n)\theta, S_1\rangle$, where $id_1 \sqsubset id_0$, iff there exists a derivation step $(A \leftarrow A) \xrightarrow{\theta} (A \leftarrow B_1, \ldots, B_n)\theta$ in the partial derivation tree of G;
- there is an edge from node $\langle id_2, A \leftarrow B_1, \ldots, B_n, S_2\rangle$ to node $\langle id_3, (A \leftarrow B_2, \ldots, B_n)\theta, S_3\rangle$, where $id_2 \sqsubset id_0$ and $id_3 \sqsubset id_0$, iff there exists a derivation step $(A \leftarrow B_1, \ldots, B_n) \xrightarrow{\theta} (A \leftarrow B_2, \ldots, B_n)\theta$ 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 10
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 evaluation 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 evaluation 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 of G which are involved in loop id, i.e., the number of nodes in $Tree$ with status $loop(ID)$ such that $id \in ID$. The counter 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. When G is completely evaluated, the fields of its table are erased, but the answers of G are maintained to speed up the evaluation of future requests for G.

Procedures. To initiate the evaluation of a goal G, a principal a generates a unique sequence of characters id_0 and sends a request $\langle id_0, a, G\rangle$ to the principal defining G. A response $\langle id_0, Ans, disposed, \{\}\rangle$ is returned to a when the evaluation of G terminates. GEM computes the answers of G (defined in policy P_G) using the following procedures:

- Process Request: if the request is not a lower request, invokes Create Table to initiate the evaluation of G. Otherwise, it sends a loop notification to the requester;
- Create Table: creates a table for G and initializes its evaluation tree with the applicable clauses in P_G;
- Activate Node: activates a *new* node in the evaluation tree of G;
- Process Response: processes the answers received for a subgoal of G;
- Generate Response: determines the requesters of G to whom a response must be sent. It is invoked when there are no more nodes in the evaluation tree of G to activate;
- Send Response: sends the computed answers of G to the requesters of G;
- Terminate: disposes the table of G. It is invoked when G is completely evaluated.

Each principal in the trust management system runs a listener that waits for incoming requests and responses. Whenever a request is received, the listener invokes

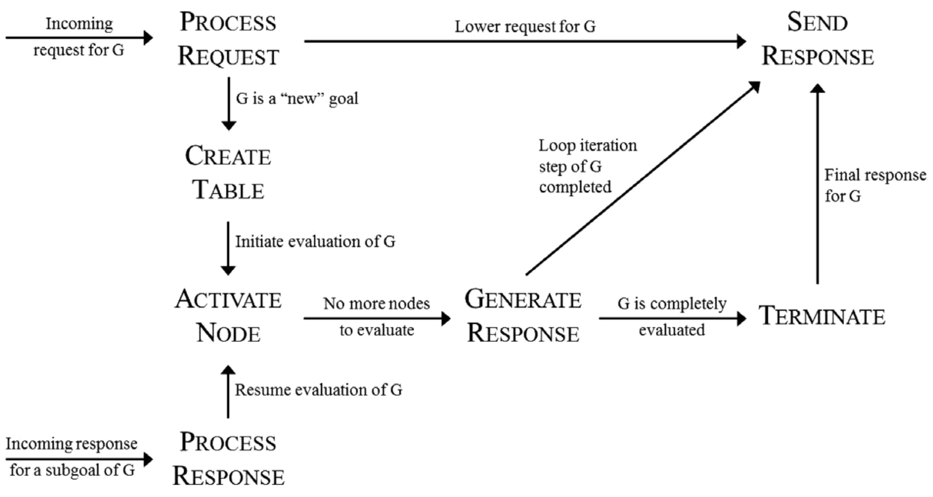

Fig. 6. Interaction among the procedures for the evaluation of a goal G.

Algorithm 1: PROCESS REQUEST

```
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($\langle id_0, req, G''\rangle$)
11 else
12     CREATE TABLE($\langle id_0, req, G\rangle$)
```

PROCESS REQUEST. Similarly, PROCESS RESPONSE is invoked upon receiving a response to a previously issued request. The interactions and dependencies among the different procedures are shown in Figure 6.

PROCESS REQUEST (Algorithm 1) takes as input a request $\langle id_0, req, G\rangle$ and, if there exists no table for a variant of G, invokes procedure CREATE TABLE to create a table for G (lines 11–12). 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). A response with 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 a 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, initiating the loop processing phase.

Algorithm 2: Create Table

```
input: a request $\langle id_0, req, G = \leftarrow A\rangle$
1 create $Table(G)$
2 initialize $Table(G)$ to $\langle\langle id_0, req, G\rangle, \{\}, \{\}, \{\}, \langle id_0, A \leftarrow A, new\rangle\rangle$
3 foreach clause $H \leftarrow B_1, \ldots, B_n$ applicable to $G$ in the local policy do
4     let $H' \leftarrow B'_1, \ldots, B'_n$ be a variable renaming of the clause s.t. it is variable disjoint from $A$, and
      $\theta = mgu(A, H')$
5     let $s$ be a unique sequence of characters
6     add subnode $\langle id_0 s, (H' \leftarrow B'_1, \ldots, B'_n)\theta, new\rangle$ to the root
7 end
8 Activate Node($G$)
```

Algorithm 3: Activate Node

```
input: a goal $G = \leftarrow A$
1  let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
2  if ($\nexists$ a non-root node $t \in T$ with status new) or ($\langle A, ID\rangle \in AS$) then
3      Generate Response($G$)
4  else
5      let $S_{root}$ be the status of the root node of $T$
6      if $S_{root} = new$ then
7          $S_{root} := active$
8      select the leftmost non-root node $t = \langle id_1, H \leftarrow B_1, \ldots, B_n, new\rangle$ from $T$
9      if $n = 0$ then
10         set the status of $t$ to answer
11         if $H$ is not subsumed by any answer in $AS$ then
12             $AS := AS \cup \{\langle H, \{\}\rangle\}$
13         Activate Node($G$)
14     else
15         if the location of $B_1$ is not ground then
16             halt with an error message  /* floundering */
17         else
18             set the status of $t$ to active
19             send request $\langle id_1, local, B_1\rangle$ to the location of $B_1$
```

3. *The request is a side request or originates from a different initial request* (lines 8–10). We treat the request as a new request; accordingly, a new table for G is created by invoking Create Table.

Create Table (Algorithm 2) inputs a request $\langle id_0, req, G\rangle$ and creates a table for goal G with HR set to $\langle id_0, req, G\rangle$, and *Tree* initialized with the clauses in the local policy applicable to G (renamed so that they share no variable with G) (lines 1–7). The identifiers of the subnodes of the root are obtained by concatenating id_0 with a unique sequence of characters s. When the initialization of the table of G is terminated, Activate Node is invoked (line 8).

Activate Node (Algorithm 3) activates a *new* node from the evaluation tree of a goal G. First, it sets the status of the root node of the evaluation tree T of goal G to *active* (lines 5–7). Then, a node with status *new* is selected from T (line 8). 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 9–13). The answer subsumption check (line 11) is important to avoid sending the same answers of a goal more than once. If the clause is not a fact, the leftmost body atom B_1 of the node's clause is selected for evaluation. In case that

Algorithm 4: 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 $Req$
```

Algorithm 5: Process Response

```
input: a response $\langle id_0, Ans, S_{ans}, Loops\rangle$
let $t = \langle id_0, H \leftarrow B_1, \ldots, B_n, S_t\rangle$ be the node in the evaluation tree of goal $G = \leftarrow A$ to which the response refers
let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
let $\langle id_1, A \leftarrow A, S_{root}\rangle$ be the root node of $T$
if $S_{root} \neq disposed$ then
    if $S_{ans} = disposed$ then
        if $S_t = loop(ID)$ then
            dispose all the nodes in $T$ involved in any loop
        $S_t := disposed$
    else
        if $S_t = loop(ID)$ then
            $ID := ID \cup Loops$
        else if $Loops \neq \{\}$ then
            $S_t := loop(Loops)$
        $AG := AG \cup \{\langle id_2, 0\rangle | id_2 \in Loops$ 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 unique sequence of characters
        add subnode $\langle id_1 s, (H \leftarrow B_2, \ldots, B_n)\theta, new\rangle$ of $t$
    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($G$)
```

the location parameter of B_1 is not ground, an error is raised and the computation is aborted by *floundering* (lines 15–16). Otherwise, a request for B_1 is sent to the corresponding location; the node identifier is used as request identifier (lines 17–19). 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 4) 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 (lines 3–7).

Response messages are processed by Process Response (Algorithm 5). The node t to which the response refers is identified by looking at the response identifier (line 1). If the status of the response is *disposed*, the selected atom B_1 of t is completely evaluated. Therefore, t 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

Algorithm 6: GENERATE RESPONSE

```
input: a goal $G = \leftarrow A$
1  let $Table(G)$ be $\langle HR, LR, AG, AS, T\rangle$
2  if ( $\nexists\langle id_0,c,loop(ID)\rangle \in T$ ) then
3      TERMINATE($G$)
4  else
5      let $\langle id_1, A \leftarrow A, S_{root}\rangle$ be the root node of $T$
6      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
7          set the counter of $id_1$ in $AG$ to the number of subgoals in $T$ involved in loop $id_1$
8          if $S_{root} = loop(ID_1)$ then
9              $S_{root} := loop(ID_1 \cup \{id_1\})$
10         else
11             $S_{root} := loop(\{id_1\})$
12         foreach $\langle id_2, req, G\rangle \in LR$ do
13             SEND RESPONSE($\langle id_2, req, G\rangle, loop(id_1), \{\}$)
14         end
15     else if $G$ is the leader of the SCC then
16         TERMINATE($G$)
17     else
18         let $Loops$ be the set $\{id_3 | \langle id_3, C\rangle \in AG$ and $id_1 \sqsubset id_3\}$
19         set the counter of each $id_3 \in Loops$ to the number of subgoals in $T$ in loop $id_3$
20         if $S_{root} = loop(ID_1)$ and $\exists id_4 \in ID_1$ s.t. $id_1 \sqsubset id_4$ then
21             SEND RESPONSE($HR, loop(id_4), Loops$)
22         else
23             SEND RESPONSE($HR, active, Loops$)
24         $S_{root} := active$
```

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 t is updated depending on whether the response contains a loop notification, i.e., set *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 t 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 unique sequence of characters s. When all answers have been processed, if the principal is not waiting for a response for any subgoal in the evaluation tree of G, ACTIVATE NODE is invoked to proceed with the evaluation of G (line 25).

GENERATE RESPONSE (Algorithm 6) is invoked when all the clauses in the evaluation 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:

1. If set LR is not empty, then goal G is the coordinator of a loop id_1, where id_1 is the identifier of the higher request for G. If there are new answers of G that have not yet been sent to the lower requests in LR, a response with status $loop(id_1)$

Algorithm 7: 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 $HR := null$
7 $LR := AG := \{\}$
```

is sent to each of them (lines 6–14). This corresponds to starting a new loop iteration for loop id_1. The status of the root node of the evaluation tree T is updated to keep track of the loops that are currently being processed (lines 8–11) and the counter of id_1 in the set of active goals AG is set to the number of subgoals in T involved in loop id_1 (i.e., the number of nodes with status $loop(ID)$ such that $id_1 \in ID$, line 7). This number corresponds to the number of subgoals for which a response in the context of loop id_1 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). G is the leader of the SCC if the only loop identifier in set AG is the identifier of the higher request for G.
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 currently being processed (lines 20–23).

Terminate (Algorithm 7) is responsible of disposing a table once all the answers of its goal G have been computed. More precisely, all the table fields are erased except for the set AS of answers of G, which are kept in case of future requests for goal G. A response with status *disposed* is sent to the requesters of HR and LR (lines 3–5).

An example of execution of GEM is in the online appendix of the paper (Appendix B).

4 Properties of GEM

This section presents the soundness, completeness, and termination results of GEM. Moreover, we discuss what information is disclosed by GEM during the evaluation of a goal.

4.1 Soundness, completeness, and termination

Here, we refer to an arbitrary but fixed set $P_1, \ldots, P_n$ of policies, and to the corresponding global policy $P = P_1 \cup \ldots \cup P_n$. To prove its soundness and completeness, we demonstrate that GEM computes a solution if and only if such a solution can be derived via SLD resolution, which has been proved sound and complete (Apt

1990). The proofs of the theorems presented in this section are provided in the online appendix of the paper (Appendix A).

The following theorem states that each solution computed by GEM can also be derived via SLD resolution using the global policy P, and is thus correct. Intuitively, this is due to the fact that the solutions generated by the algorithm are obtained using the clause resolution mechanism, which produces correct results.

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. $P = P_1 \cup \ldots \cup P_n$). 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 $G_i\sigma$.

Next, we present the completeness result.

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. $P = P_1 \cup \ldots \cup P_n$). Assume that running GEM on G_1 (w.r.t. $P = P_1 \cup \ldots \cup P_n$) 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 $G_1\sigma$.

Finally, we state that GEM always terminates.

Theorem 3 (*Termination*)
Given a goal G evaluated with respect to a finite global policy P, GEM terminates.

4.2 Disclosed information

A primary objective of GEM is to preserve the confidentiality of intensional policies. Here, we discuss what policy information principals are able to collect during the evaluation of a goal, and classify GEM according to the confidentiality levels defined in Section 2.

First, let us define the following notation. Let P be a global policy, and G_a and G_b be two goals in P defined by principals a and b respectively. We say that goal G_a *depends* on goal G_b if there is a path from G_a to G_b in the call graph of the evaluation of G_a with respect to P. Since each edge in the call graph represents a request in GEM, and in trust management each request corresponds to a delegation of authority, if G_a depends on G_b then we say that there is a *chain of trust* from principal a to principal b.

We also introduce some notation on request identifiers. As mentioned in Section 3.2, the identifiers in an identifier domain can be defined in several ways (e.g., applying a hash function to the identifier of a higher request). In this paper, we have considered an identifier domain where identifiers are obtained by concatenating the identifier of a higher request with a sequence of characters. Here, we discuss what information is disclosed by GEM during goal evaluation using this identifier domain. We classify identifiers obtained by concatenation according to two dimensions:

traceability and *length*. Given two request identifiers id_1 and id_2 for goals G_1 and G_2 respectively, such that $id_2 \sqsubset id_1$, the traceability dimension refers to the ability of a principal to infer which principals are involved in the evaluation of the goals in the path from G_1 to G_2. On the other hand, the length dimension defines the ability to determine the number of goals in the path from G_1 to G_2. Let $id_0 s_1 \cdots s_n$ be a request identifier, where id_0 is the identifier of the initial request and each s_i (for $i \in \{1, \ldots, n\}$) is a sequence of characters added by a principal p_i to the identifier $id_0 s_1 \cdots s_{i-1}$ of a higher request. For what concerns the traceability dimension, we say that $id_0 s_1 \cdots s_n$ is a *traceable* identifier if each string s_i uniquely identifies (the location of) principal p_i (as done, for instance, by the identifiers in the example in Section 3.2); otherwise, we say that identifier $id_0 s_1 \cdots s_n$ is *untraceable*. The length dimension is defined based on the number of characters concatenated by each principal p_i to the request identifier $id_0 s_1 \cdots s_{i-1}$. Let $len(s_i)$ denote the number of characters in the string s_i. If $len(s_1) = \ldots = len(s_n)$, then we say that $id_0 s_1 \cdots s_n$ is a *fixed-length* identifier; otherwise, we say that $id_0 s_1 \cdots s_n$ is a *variable-length* identifier (we assume that cryptographic techniques are in place to avoid collision of identifiers; Hoch and Shamir 2008). Note that a traceable identifier does not necessarily disclose information about the number of goals in the path between two goals; this is because a goal defined by a principal a can have several subgoals defined in a's policy. Consider, for instance, the traceable request identifier h:12c1:345, obtained by concatenating a request identifier with a principal's identifier and a variable-length sequence of digits for each goal evaluated by the principal. Even though identifier h:12c1:345 shows that the principals involved in the computation are hospital h and company $c1$, it does not confer information about the number of goals evaluated by those principals. Company $c1$, for example, might have evaluated two locally defined goals, concatenating the higher request h:12 received from hospital h first with its identifier $c1$ and digit "3" (separated by a semicolon), and then with the sequence of digits "45."

We are now ready to present what information a principal b is able to learn about the local policy of a principal a where a goal G_a is defined. First of all, by requesting the evaluation of G_a, b learns the set of answers to the request, i.e., the extensional policy relative to G_a; this is necessary for any goal evaluation algorithm. As mentioned in Section 1, the confidentiality of extensional policies can be protected, for instance, by relying on hidden credentials (Bradshaw *et al.* 2004; Frikken *et al.* 2006) or trust negotiation algorithms (Winsborough *et al.* 2000; Winslett 2003). Here, we are more interested in what b can learn about the intensional policy defining G_a.

By sending a request for G_a (say, with identifier id_1), b can learn whether G_a depends on some goal G_b defined in her policy. Indeed, if b receives a request for G_b with identifier id_2 such that $id_2 \sqsubset id_1$, then b knows that G_a depends on G_b.

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

- what are the goals $G_{b_1}, \ldots, G_{b_n}$ defined in her policy on which G_a depends;
- for each $G_{b_i} \in \{G_{b_1}, \ldots, G_{b_n}\}$, b knows who is the principal p_i that requested G_{b_i}; therefore, b learns that G_a depends on a goal defined by p_i, i.e., that there

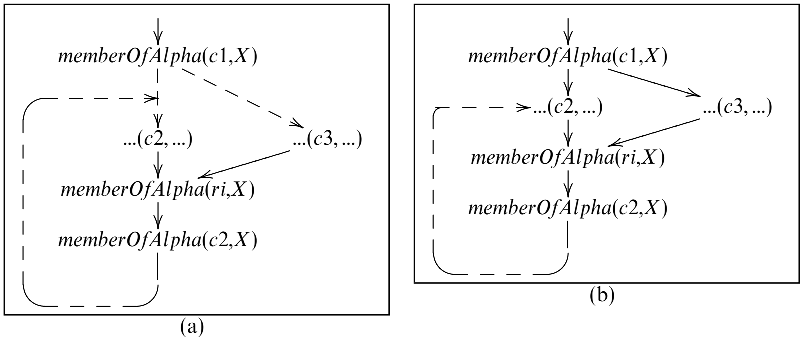

Fig. 7. Part of the call graph that can be inferred by *ri* using (a) variable-length, untraceable request identifiers and (b) fixed-length request identifiers.

is a chain of trust from a to p_i (and from p_i to b). Principal b, however, does not necessarily learn which is the goal defined by p_i on which G_a depends;

- depending on how the identifiers are constructed, b might be able to learn additional information about the path from G_a to G_{b_i}. In particular, if the identifier of the request for G_{b_i} is fixed-length, b is able to infer the number of goals in the path from G_a to G_{b_i}. Additionally, if the identifier is traceable, b also learns who are the principals defining those goals.

Thus, GEM can be classified as E1-I1 according to the classification criteria in Section 2. In fact, principals learn all the answers of a goal along with some dependencies among the goals involved in an evaluation.

We now illustrate the concepts presented above with an example, using the global policy introduced in Section 3.2 and the call graph shown in Figure 5(a) (ignore, for now, the request identifiers depicted in the figure). Assume that the research institute *ri* requests goal *memberOfAlpha(c1,X)* to $c1$. If the identifiers used in the computation were variable-length and untraceable, *ri* would be able to learn that:

- *memberOfAlpha(c1,X)* depends on goal *memberOfAlpha(ri,X)* defined in its policy;
- *memberOfAlpha(c1,X)* depends on *some* goal G_{C2} defined in $c2$'s policy and on *some* goal G_{C3} defined by $c3$; however, it does not learn which goals they are. Furthermore, due to the loop notification received from $c2$ following the evaluation of goal *memberOfAlpha(c2,X)*, *ri* learns that *memberOfAlpha(c2,X)* depends on *some* goal in the path from *memberOfAlpha(c1,X)* to G_{C2}.

Figure 7(a) represents *ri*'s "view" of Figure 5(a), that is, the part of the call graph that *ri* can infer. In the graph, we denote with dots ("...") the predicate symbols and terms that *ri* does not learn; a dashed edge from a goal G_1 to a goal G_2 indicates that *ri* is able to infer that G_1 depends on G_2, but not the (number of) goals and principals in the path from G_1 to G_2. Since *ri* does not learn that G_{C2} is actually goal *memberOfAlpha(c2,X)*, with respect to the call graph in Figure 5(a), in Figure 7(a) goal *memberOfAlpha(c2,X)* is "duplicated." The reason why *ri* is not able to infer that G_{C2} is *memberOfAlpha(c2,X)*, and even more, does not learn whether *memberOfAlpha(c2,X)* is in the path from *memberOfAlpha(c1,X)* to *memberOfAlpha(ri,X)* or is a lower goal, is that the only information that *ri* receives in response to the request for *memberOfAlpha(c2,X)* (besides the answers of the goal) is a notification that *memberOfAlpha(c2,X)* is in a loop, say with identifier id_l. *ri* can observe that id_l corresponds to a request higher than the request for *memberOfAlpha(ri,X)*, i.e., that the loop coordinator is higher than *memberOfAlpha(ri,X)*. However, *ri* cannot infer whether the loop was formed by its own request (in which case *ri* would learn that G_{C2} is *memberOfAlpha(c2,X)*), or by a request issued by *c*2 when evaluating *memberOfAlpha(c2,X)*, or even by the evaluation of a goal on which *memberOfAlpha(c2,X)* depends. In other words, because of the variable-length and untraceable nature of identifiers, *ri* does not know the number of goals in the path from *memberOfAlpha(ri,X)* to the loop coordinator.

In addition to the information above, if the identifiers used in the computations were fixed-length, *ri* would also be able to learn that:

- one of the paths from *memberOfAlpha(c1,X)* to *memberOfAlpha(ri,X)* consists of *three* goals: *memberOfAlpha(c1,X)*, G_{C2}, and *memberOfAlpha(ri,X)*. Furthermore, *ri* can infer that G_{C2} is the coordinator of loop id_l. However, *ri* still does not learn whether G_{C2} is *memberOfAlpha(c2,X)*;
- the other path from *memberOfAlpha(c1,X)* to *memberOfAlpha(ri,X)* consists of *three* goals: *memberOfAlpha(c1,X)*, G_{C3}, and *memberOfAlpha(ri,X)*.

The part of the call graph that *ri* can infer in a computation with fixed-length identifiers is shown in Figure 7(b). Note that, in this example, the information that *ri* can infer is the same independently from the traceability of the identifiers. This is because *ri* already knows all the principals in the paths from *memberOfAlpha(c1,X)* to *memberOfAlpha(ri,X)*: *c*1 is the principal to whom *ri* sent the initial request, and *c*2 and *c*3 are the principals from whom *ri* received the request for *memberOfAlpha(ri,X)*. Even with traceable identifiers, *ri* would not be able to infer more information about the path from *memberOfAlpha(c2,X)* to G_{C2}, as the only information received by *ri* from *c*2 is the loop identifier.

A principal might attempt to infer a bigger portion of the call graph by issuing requests for each goal defined by the principals in the trust management system. For instance, *ri* can infer more information by issuing a request for each goal defined by companies *c*1, *c*2, and *c*3. In particular, by issuing a request for *memberOfAlpha(c2,X)*, in a computation with fixed-length and traceable identifiers *ri* would learn that *memberOf-Alpha(c2,X)* is the goal defined by *c*2 that depends on *memberOfAlpha(ri,X)* [Fig. 8(a)]. By also issuing a request for *memberOfAlpha(c3,X)*, *ri* could infer the whole call graph [Fig. 8(b)]. Note, however, that the global policy considered here is a relatively simple policy with few goals and principals. A more complex policy would complicate and sometimes prevent the inference of goal dependencies. Moreover, some information about the global policy would not be deducible by *ri* when using variable-length and untraceable identifiers. All the edges in the call graph in Figure 8(b), for instance, would be dashed edges if variable-length untraceable identifiers were used. Finally, it is worth noting that even though *ri* might be able to learn the whole call graph, that graph might correspond to different intensional policies (Costantini 2001). For example, *ri*

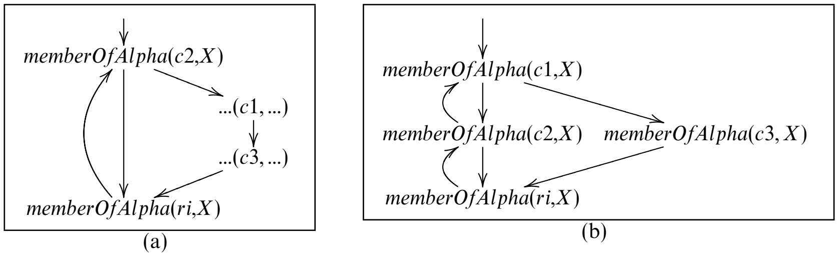

Fig. 8. Part of the call graph that can be inferred by *ri* by (a) requesting goal *memberOfAlpha(c2,X)* and (b) issuing a request for the goals defined by each principal in the trust management system.

is not able to learn whether *memberOfAlpha(c2,X)* and *memberOfAlpha(c3,X)* are connected by disjunction or conjunction in *c*1's policy.

To conclude, we argue that when using an appropriate identifier domain the knowledge about goal dependencies disclosed by GEM is not sufficient for a principal *b* to infer the intensional policy relative to a goal G_a defined by a principal *a*. Principal *b* always learns whether G_a depends on goals defined in her policy, but most likely not all the goals in the global policy on which G_a depends. Consider, for instance, clause 2 on page 2. A principal other than *c*1 and *mc* cannot learn that *memberOfAlpha(c1,X)* depends on *projectPartner(mc,Y)*. We believe that the information that *b* can infer is consistent with the concept of trust management. In fact, if G_a depends on a goal defined by *b*, then there is a chain of trust from *a* to *b*; it seems legitimate that the existence of such a chain may not remain secret to *b*.

5 Practical evaluation

We implemented the algorithms presented in Section 3.3 in Java and conducted several experiments to evaluate the performance of GEM. In particular, we first tested the implementation with the example policies defined in Section 3.2 and in the online appendix of the paper (Appendix B). Then, we modified those policies to assess the scalability of GEM with respect to an increase in the number of principals in the trust management system and the number of clauses in the global policy.

We carried out the experiments by running GEM on four machines located in different area networks. More precisely, we employed two machines located within the Eindhoven University of Technology (TU/e) network, and two located at the University of Twente (UT). The two TU/e machines mount an Intel Core 2 Quad 2.4 GHz processor with 3 GB of RAM and a 32 bit Windows operating system. The UT machines are 32 bit Ubuntu machines with the same processor but 2 GB of RAM. In each experiment, we have assigned approximately one-fourth of the principals (i.e., one-fourth of the local policies) in the global policy to each machine. The exchange of messages between principals on different machines is via HTTP (`javax.servlet.http` Servlet API).

To present the results of the experiments, we group them into two sets. The first set of experiments (Section 5.1) studies the performance of GEM for an increasing number of principals, clauses, and loops in the global policy. The second set (Section 5.2) shows, for some of the global policies in the first set, the effects of increasing the size of the extensional policy (i.e., the number of facts in the global policy). For each experiment we report the following results: the number of principals involved in the computation (denoted by *Princ*); the number of tables created by GEM during the computation (*Tab*); the number of clauses evaluated (*Clauses*); the sum of the computation times on the four machines, expressed in milliseconds (*CTime*); the total time (*TTime*), expressed in milliseconds, given by the *CTime* plus network communication time; the total memory occupied by GEM on the four machines, expressed in kilobytes (*TMem*); the maximum memory occupied by the tables of the goals created by GEM on the four machines, expressed in kilobytes (*TabMem*); the memory occupied by the tables of the goals created by GEM on the four machines after the tables' disposal, expressed in kilobytes (*EndTabMem*); the number of requests issued during the computation (*Req*); the number of loops identified during the computation (*Loops*); the number of response messages exchanged between principals (*Resp*); the number of non-empty response messages (i.e., response messages containing at least one answer) exchanged between principals (*Resp&Ans*); the total number of answers computed by GEM during the evaluation of the policy (*Ans*).

5.1 Experiments set 1: increasing the number of principals, clauses, and loops

In the first set of experiments, we conducted three groups of (sub)experiments to evaluate the performance of GEM in response to an increase in (1) the number of principals and clauses, (2) the number of loops, and (3) both the number of principals, clauses, and loops in a global policy. To evaluate GEM in response to an increase in the number of principals and clauses (experiments group 1), we created six variants of the global policy in the online appendix of the paper (Appendix B). For the second group of experiments, we created six variant of the global policy defined in Section 3.2. Similarly, other six variants of the same policy were created for the experiments in the third group, in order to increase the number of both principals, clauses, and loops. The call graphs of the six variants of the global policies are shown in Figure C 1 of the online appendix of the paper. Each variant is denoted by an identifier that goes from x.0 to x.5 (where x is either 1, 2, or 3 depending on the experiment group for which they are used), where variant x.0 represents the original policy. Note that for the sake of compactness, Figures C 1(b) and 1(c) show the folded graph of the global policies, i.e., they do not represent the reevaluation of goals due to side requests. Since in GEM the computation is based on the unfolded versions of the graph [e.g., the graph in Figure 5(b) for variants 2.0 and 3.0], the number of lower requests occurring in the actual computation is higher than the one displayed in the figures. For instance, the number of lower requests for the leader of the SCC goes up to seven in variants (and hence experiments) 2.5 and 3.5.

Table 1 presents the results of the three groups of experiments. Each row in the table shows the results for the variant of the global policy with identifier indicated in column *ID*. The total time (*TTime*) and the computation time (*CTime*) are

Table 1. *Performance evaluation results: experiments set 1*

ID	Princ	Tab	Clauses	TTime (CTime) in ms	TMem in kB	TabMem (EndTabMem) in kB	Req	Loops	Resp (Resp&Ans)	Ans
1.0	4	4	6	286.1 (6.3)	32	18 (15)	5	1	9 (6)	9
1.1	7	7	12	305.9 (8.5)	53	39 (27)	9	2	17 (11)	26
1.2	10	10	18	315.2 (11.2)	78	66 (44)	13	3	25 (17)	49
1.3	13	13	24	322.8 (14.7)	108	97 (63)	17	4	33 (22)	78
1.4	16	16	30	327.0 (17.0)	140	134 (85)	21	5	41 (27)	113
1.5	19	19	36	331.7 (19.8)	180	175 (108)	25	6	49 (33)	154
2.0	4	6	8	830.3 (10.7)	47	37 (32)	10	4	31 (17)	20
2.1	5	10	10	1037.2 (13.0)	68	61 (50)	16	6	48 (19)	32
2.2	6	15	12	1283.8 (16.5)	92	91 (74)	23	8	72 (30)	46
2.3	7	21	14	1459.9 (19.9)	124	122 (100)	31	10	95 (34)	62
2.4	8	28	16	1673.6 (23.4)	163	153 (127)	40	12	125 (47)	80
2.5	9	36	18	1840.0 (27.2)	206	189 (159)	50	14	154 (53)	100
3.0	4	6	8	830.3 (10.7)	47	37 (32)	10	4	31 (17)	20
3.1	7	16	16	1653.6 (23.9)	151	149 (110)	28	12	113 (71)	112
3.2	10	36	24	3029.0 (45.2)	454	427 (297)	64	28	309 (199)	384
3.3	13	76	32	5672.4 (78.8)	1210	1133 (752)	136	60	773 (535)	1088
3.4	16	156	40	10479.7 (134.4)	3024	2827 (1824)	280	124	1789 (1276)	2800
3.5	19	316	48	21939.5 (310.2)	7285	6862 (4267)	568	252	4569 (3477)	6816

the average times for 100 runs of each experiment; the values in all the other columns are constant for every run, as they depend on the structure of the global policy.

Before interpreting the results of the experiments, let us provide some general comments on the relationship between the values in different columns of Table 1. First, in every experiment the number of requests (*Req*) is equal to the number of tables generated (*Tab*) plus the number of loops identified by GEM (*Loops*); this is because a request is either a higher request, which leads to the creation of a table, or a lower request, and thus forms a loop. Second, the number of response messages (*Resp*) is always at least as large as the number of requests, since to every request is given a response, even if with an empty set of answers. The number of empty response messages (i.e., response messages with an empty set of answers) can be observed by subtracting the number of response messages containing at least one answer (*Resp&Ans*) from the total number of response messages *Resp*. Finally, for some experiments (namely in the computation of variants 2.2 to 2.5 and 3.2 to 3.5), the number of tables generated by GEM is higher than the number of clauses (*Clauses*) and thus the number of goals defined in the global policy. This is because while column *Tab* reports the *total* number of tables generated during a computation, column *Clauses* shows the number of *different* clauses being evaluated. In other words, we do not count twice the clauses used for the evaluation of a goal that is reevaluated because of a side request. The reason behind this choice is that, on the one hand, we are interested in showing the impact of the number of tables generated and answers computed (*Ans*) during the evaluation of a policy on the memory usage (*TMem* and *TabMem*); on the other hand, considering the number of different clauses gives a better insight on the size of a policy.

A first interesting outcome of the experiments is that the time and memory results in Table 1 increase approximately linearly with the number of loops in the global policy (see Fig. C 2 of the online appendix of the paper for a graphical overview). When the number of loops is low (experiments in groups 1 and 2), the time and memory usage are negligible, but as the number of loops increases considerably (experiments group 3), the *TTime*, *TMem*, and *TabMem* get substantially higher. This is because an increase in the number of loops [especially if nested, as in the global policy in Fig. C 1(b) of the online appendix of the paper] leads to an increase in the number of response messages, answers, and (since GEM reevaluates side requests) tables generated in a computation.

Another interesting result is represented by the difference between total time and computation time [see Fig. C 2(a) of the online appendix]. In fact, the *TTime* ranges from 16.7 times the *CTime* for policy variant 1.5, up to 80.1 times the *CTime* for variant 2.1. This implies that most of the *TTime* is devoted to network communication. Therefore, we can conclude that the larger the number of requests and response messages and the number of answers per response message, the higher is the difference between *TTime* and *CTime*. Furthermore, we point out that in a real distributed system where to each principal corresponds a different machine (possibly in a different area network) the *TTime* would be much higher than the results in Table 1, especially when the number of principals increases.

Finally, Table 1 shows that the number of answers in a computation is always larger than the number of response messages containing at least one answer; in other words, each message in *Resp&Ans* carries more than one answer on average. This information, combined with the observation on the difference between *TTime* and *CTime*, suggests that GEM can consistently reduce network overhead with respect to other goal evaluation algorithms which send one message for each computed answer (e.g., Alves *et al.* 2006), especially when the number of principals and facts in the global policy increases.

5.2 Experiments set 2: Increasing the size of extensional policies

In a real-world policy, we expect the size of the extensional policy (i.e., the number of facts that can be derived from the policy) to substantially exceed the size of the intensional policy (i.e., the number of clauses used to derive new facts). Consider, for example, the students of a university: while there are only a few rules that define the procedure for becoming a student, the number of students usually goes beyond several thousands. The goal of the experiments presented in this section is to evaluate the impact on the performance of GEM of an increase in the number of facts in a global policy. To this end, we considered some of the global policies introduced in Section 5.1 and increased the size of their extensional policies by a factor of 10, 50, and 100; in particular, we modified variants 1.0, 1.5, 2.0, 2.5, and 3.2. We could not perform experiments on variants 3.3–3.5 due to the limited memory available on some of the machines used in the experiments.

Table 2 shows the results of the second set of experiments. In the table, suffix "a" on a variant's identifier indicates an increase by a factor of 10 of the number of facts in that variant of the global policy, suffix "b" indicates an increase by a factor of 50, and suffix "c" indicates an increase by a factor of 100. Note that the number of answers (*Ans*) computed on variants 1.0 and 1.5 grows less than a factor of 10, 50, and 100 because, in order to not modify the structure of the global policy, the number of facts in the policies of principals *mc*1 to *mc*6 in Fig. C 1(a) of the online appendix of the paper was not increased; more precisely, those policies always consist of only two facts. Similar to the previous experiments, *TTime* and *CTime* are the average times for 100 runs of each experiment.

The results in Table 2 show that memory usage (*TMem* and *TabMem*) and computation time grows faster than the other values for policies with a very large number of computed answers (i.e., variants 1.5c, 2.5c, and 3.2c). For what concerns memory usage [Fig. C 3(b) of the online appendix of the paper], the extra overhead is due to the accompanying increase of the information that needs to be stored in tables (i.e., clauses, loops, and answers). After the disposal of the tables employed in the computation, there is a decrease of up to 38% of the memory usage (*EndTabMem*). This suggests that it is very important to delete as much information as possible from the table of a goal when the goal is completely evaluated, as this leads to a substantial reduction of memory usage. In this respect, GEM has the advantage of enabling principals to detect when the evaluation of the single goals involved in a computation is completed, and immediately clean up the table of those goals.

Table 2. *Performance evaluation results: experiments set 2*

ID	Princ	Tab	Clauses	TTime (CTime) in ms	TMem in kB	TabMem (EndTabMem) in kB	Req	Loops	Resp (Resp&Ans)	Ans
1.0	4	4	6	286.1 (6.3)	32	18 (15)	5	1	9 (6)	9
1.0a	4	4	24	293.7 (13.5)	81	69 (58)	5	1	9 (6)	72
1.0b	4	4	104	334.8 (33.9)	308	300 (255)	5	1	9 (6)	352
1.0c	4	4	204	519.8 (78.8)	590	586 (498)	5	1	9 (6)	702
1.5	19	19	36	331.7 (19.8)	180	175 (108)	25	6	49 (33)	154
1.5a	19	19	144	391.6 (80.3)	1215	1210 (790)	25	6	49 (33)	1432
1.5b	19	19	624	1589.4 (817.7)	5857	5764 (3786)	25	6	49 (33)	7112
1.5c	19	19	1224	4356.5 (2986.9)	11711	11541 (7098)	25	6	49 (33)	14212
2.0	4	6	8	830.3 (10.7)	47	37 (32)	10	4	31 (17)	20
2.0a	4	6	26	843.7 (23.9)	192	184 (144)	10	4	31 (17)	200
2.0b	4	6	106	948.4 (68.6)	846	844 (651)	10	4	31 (17)	1000
2.0c	4	6	206	1441.2 (161.3)	1672	1658 (1279)	10	4	31 (17)	2000
2.5	9	36	18	1840.0 (27.2)	206	189 (159)	50	14	154 (53)	100
2.5a	9	36	36	1871.8 (58.1)	1017	976 (726)	50	14	154 (53)	1000
2.5b	9	36	116	2490.1 (259.1)	4665	4534 (3315)	50	14	154 (53)	5000
2.5c	9	36	216	3875.9 (755.2)	9225	8984 (6538)	50	14	154 (53)	10000
3.2	10	36	24	3029.0 (45.2)	454	427 (297)	64	28	309 (196)	384
3.2a	10	36	78	3142.3 (158.1)	3306	3259 (2081)	64	28	309 (196)	3840
3.2b	10	36	318	6277.8 (1789.6)	16103	15954 (10032)	64	28	309 (196)	19200
3.2c	10	36	618	13906.3 (6346.1)	31938	31883 (19968)	64	28	309 (196)	38400

Further to the increase in the number of exchanged messages, we believe the extra overhead in *TTime* and *CTime* in computations with a large number of answers [the peaks for variants 1.5c, 2.5c, and 3.2c in Fig. C 3(a) of the online appendix of the paper] to be due to the growth of the data structures used by GEM to search, for example, for new answers and clauses to be activated. In addition, contrary to experiments set 1, in this set of experiments the *TTime* is not always dominated by network communication time. In particular, for variant 1.5c the *CTime* is twice the network time, and for variant 1.5b the *CTime* is slightly larger than the network time. In the remaining experiments, the *TTime* ranges from 2.2 times the *CTime* for policy variant 3.2c, up to 77.7 times the *CTime* for variant 2.0. Moreover, note that the difference between *TTime* and *CTime* always decreases as the number of facts grows. This is due to the fact that the number of messages exchanged between principals remains constant while the size of the extensional policy is increased.

To conclude, we highlight again how the "wait" mechanism that GEM employs to collect a maximum set of answers before sending a response can contribute to reduce the network overhead, especially for global policies with a large extensional policy. For example, in the experiment on variant 3.2c, GEM sends "only" 196 response messages, while other distributed goal evaluation algorithms (e.g., Alves *et al.* 2006) would send 38,400 messages, one for each computed answer. Intuitively, the latter approach would lead to a network communication time much higher than the 7.5 seconds spent by GEM.

6 Dealing with negation

GEM is devised to work with *definite* logic programs, i.e., programs without negation. Negation is used by some trust management systems (e.g., Czenko *et al.* 2006; Dong and Dulay 2010) to express non-monotonic constraints, such as separation of duty or "distrust" in principals with certain attributes (e.g., employees of a rival company). Here, we discuss how GEM can be extended to support the use of *negation as failure*.

Negation as failure is an inference rule that derives the truth of a negated body atom $not(B)$ by the failure to derive B. The problem when allowing the use of negation (as failure) is that in the presence of *loops through negation* (i.e., loops involving negated atoms) a program may have several minimal models (Gelfond and Lifschitz 1988). For instance, program $p \leftarrow not(q)$, $q \leftarrow not(p)$ has two minimal models: $\{p\}$ and $\{q\}$. Moreover, these two models are not well-founded (Van Gelder *et al.* 1991), as there is no clause in the program demonstrating that p respectively q are true. Another undesired consequence of loops through negation is that they may introduce "inconsistencies" in a program, as shown at the end of this section. There are additional consequences of loops through negation (Van Gelder *et al.* 1991; Apt and Bol 1994), which we do not discuss further as they go beyond the scope of this paper. In fact, our goal is not to have a full-fledge handling of negation, but to allow the use of negation in policies while preventing loops through negation.

Loops through negation are a well-studied issue in the logic programming literature. There are three standard solutions to the problems they raise: (a) forbidding the presence of loops through negation, as done by the weakly perfect model semantics (Przymusinska and Przymunsinski 1990), which is defined only for *weakly stratified programs* [that include *locally stratified* (Przymusinski 1988) and *stratified* programs (Apt *et al.* 1988)]; (b) using a three-valued semantics (Przymusinski 1990), including the truth value *undefined* next to *true* and *false*, as done by the well-founded semantics (Van Gelder *et al.* 1991) and Fitting's semantics (Fitting 1985), where the semantics of p in the program $p \leftarrow not(p)$ is *undefined*; (c) following a multi-model approach, as in stable models (Gelfond and Lifschitz 1988).

Our solution follows an approach similar to (a), since solutions (b) and (c) are not suitable for trust management systems. In fact, relying on a three-valued semantics (b) requires additional mechanisms to determine whether the truth value of the goals involved in a loop through negation is true, false, or *undefined* [e.g., delaying in SLG resolution (Chen and Warren 1996)]. In trust management, however, loops through negation are inherently wrong, as they indicate conflicting policy statements issued by principals among which there is a mutual trust relationship, and thus should not be processed. Similarly, solution (c) would imply that an access request should be either granted or revoked depending on which (truth) value we "choose" to assign to the goals in a computation, which is clearly not a safe approach. Solution (a) can also not be applied straightforwardly in our context because the definition of weakly stratified program relies on a "global ordering" among *all* the (ground) atoms in a global policy. This would require principals to agree beforehand on the allowed dependencies among (ground) goals; however, in a trust management system principals often do not know each other until their first interaction. Therefore, rather than forbidding the presence of loops through negation, we prevent their evaluation. In this respect, the added value of GEM is its ability to detect loops at runtime. We exploit this feature by introducing an additional runtime check to the algorithm, which causes the computation to flounder if a loop involving a negated goal is detected. The check is safe in that if the computation does not flounder, then it always returns a correct answer.

In summary, GEM can be extended to allow the use of negation in policy statements as follows. Given a clause with a literal $not(B)$ selected for evaluation:

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

Conditions (1)–(3) are standard when defining negation as failure: (1) is necessary to guarantee correctness (Apt 1990), while (2) and (3) define the semantics of negation. Note that condition (3) also captures the case of *infinite failure*, as done, for instance, by the well-founded semantics (Van Gelder *et al.* 1991). For example, given a policy composed of clauses $q \leftarrow not(p)$ and $p \leftarrow p$, and a goal q, GEM first completely evaluates clause $p \leftarrow p$, detecting the loop and deducing that no answer of p can be derived; consequently, it concludes that q is true. Condition (4) states that the algorithm flounders if it detects a loop through negation. The "floundering message" is propagated to all the goals involved in the loop similarly to a response message, so that their evaluation is aborted.

Note that floundering due to condition (1) can be avoided by restricting to *well-moded* programs (Apt and Marchiori 1994). Different from weakly stratified programs, which require an ordering among all the goals in a global policy, the definition of well-moded program requires each clause *independently* to be well-moded. Therefore, by requiring local policies to be well-moded, this type of floundering can be prevented.

It is straightforward to demonstrate that the proposed extension of GEM:

- always terminates (for arbitrary global policies and requests), because the only difference with the standard GEM algorithm (which terminates) is the presence of an additional termination condition, and
- for non-floundering computations, it is sound and complete with respect to the stable models and well-founded semantics.

We now show how the extended algorithm deals with negation by means of an example. We consider a scenario inspired by the one introduced in Section 1, where the pharmaceutical company $c1$ needs to determine the set of principals participating to project *Alpha*. Project *Alpha* is a multidisciplinary project which requires the collaboration of experts from several fields: physicians, biologists, chemists, etc. Company $c1$ already formed a team of qualified chemists to work on the project and delegates to the partner company $c2$ the authority of determining the remaining project members. To avoid interference with the work of its trusted chemists, however, $c1$ wants to prevent chemists of $c2$ to take part to the project. In its definition of project members, $c2$ also includes the members of project *Alpha* at $c1$. This scenario can be represented by the following policy statements:

1. memberOfAlpha($c1$,X) $\leftarrow$ memberOfAlpha($c2$,X), *not*(chemist($c2$,X)).
2. memberOfAlpha($c1$,*david*).
3. chemist($c1$,*david*).
4. memberOfAlpha($c2$,X) $\leftarrow$ memberOfAlpha($c1$,X).
5. memberOfAlpha($c2$,*alice*).
6. chemist($c2$,*alice*).
7. memberOfAlpha($c2$,*eric*).

To compute the answers of goal *memberOfAlpha(*$c1$,X*)*, GEM proceeds as follows. First, clause 1 is evaluated by $c1$, leading to a request for goal *memberOfAlpha(*$c2$,X*)* to $c2$. The evaluation of the first applicable clause in $c2$'s policy (clause 4) forms a loop, identified by $c1$. The loop processing phase continues at $c2$, which identifies the first two answers of *memberOfAlpha(*$c2$,X*)* (i.e., *memberOfAlpha(*$c2$,*alice)* and *memberOfAlpha(*$c2$,*eric)*, clauses 5 and 7 resp.) and sends them to $c1$. For each of these answers, $c1$ requests to $c2$ whether the project member is a chemist. The evaluation of *chemist(*$c2$,*alice)* succeeds at $c2$ (clause 6), while *chemist(*$c2$,*eric)* fails; therefore, their negated counterpart in clause 1 fails and succeeds respectively, leading to a new answer at $c1$, namely *memberOfAlpha(*$c1$*, eric)*. This answer, together with the answer derived by evaluating clause 2 (i.e., *memberOf-Alpha(*$c1$*,david)*), is sent by $c1$ to $c2$, starting the second loop iteration. In this iteration, $c2$ finds one new answer, *memberOfAlpha(*$c2$*,david)*, which is immediately returned to $c1$. Now, $c1$ evaluates clause 1 based on the new answer received from $c2$. Since David is not a chemist at $c2$, $c1$ derives again answer *memberOfAlpha(*$c1$*,da-vid)*, which had already been computed in the previous iteration. Thus, the evaluation of *memberOfAlpha(*$c1$*,X)* terminates with two answers: *memberOfAlpha(*$c1$*,eric)* and *memberOfAlpha(*$c1$*,david)*.

The example above shows that GEM can easily support policies including both loops and negation. We now show how GEM operates in presence of loops through negation. Consider the following policy statements complementing the global policy above:

8. chemist($c2$,X) ← memberOfAlpha($c1$,X), chemist($c3$,X).
9. chemist($c3$,*eric*).

Clause 8 states that all the members of project *Alpha* at $c1$ that work as chemists at the other partner company $c3$ are also chemists at $c2$. Note that clause 8 is “inconsistent” with clause 1. In fact, clause 1 defines as members of project *Alpha* principals that *are not* chemists at company $c2$; at the same time, clause 8 states that chemists at $c2$ *are* members of project *Alpha* at $c1$. For this reason, when $c1$ ascertains that goals *memberOfAlpha(*$c1$*,eric)* and *chemist(*$c2$*,eric)* are in a loop, it raises an error and the computation flounders. In fact, in the example computation above, the evaluation of *chemist(*$c2$*,eric)* by $c2$ would lead to a contradiction: if Eric were not a chemist at $c2$, he would be a member of project *Alpha*; however, if Eric were a member of project *Alpha*, he would be a chemist at $c2$.

7 Related work

Research on goal evaluation has been carried out in the field of both logic programming and trust management. In this section we compare our work with existing frameworks focusing on the information disclosed during the evaluation process, based on the classification criteria defined in Section 3.1. Additionally, we indicate whether the analyzed systems employ a centralized or distributed goal evaluation strategy and discuss the termination detection mechanism they adopt. 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 3 summarizes the results of this analysis. In the table, LP denotes the algorithms proposed in the logic programming domain, while TM denotes trust management systems.

SLG resolution (Chen and Warren 1996), TP resolution (Shen *et al.* 2001), DRA (Guo and Gupta 2001), OPTYap (Rocha *et al.* 2005), and the work by Hulin (1989) are centralized tabling systems in which the complete program (i.e., the global policy) is available during the evaluation. Therefore, these five systems are

Table 3. *Comparison between goal evaluation algorithms*

	Frameworks	Evaluation	Computation termination	Goal termination	Classification
LP	SLG (Chen and Warren 1996)	Centralized	Centralized	Centralized	E1-I3
	TP resolution (Shen *et al.* 2001)	Centralized	Centralized	Centralized	E1-I3
	DRA (Guo and Gupta 2001)	Centralized	Centralized	Centralized	E1-I3
	OPTYap (Rocha *et al.* 2005)	Centralized	Centralized	Centralized	E1-I3
	Hulin (1989)	Centralized	Centralized	Centralized	E1-I3
	Damásio (2000)	Distributed	Distributed	Distributed	E1-I2
	Hu (1997)	Distributed	Distributed	Distributed	E1-I2
TM	RT (Li *et al.* 2003)	Centralized	Centralized	Centralized	E1-I3
	Tulip (Czenko and Etalle 2007)	Centralized	Centralized	Centralized	E1-I3
	SecPAL (Becker *et al.* 2010)	Centralized	Centralized	Centralized	E1-I3
	SD3 (Jim and Suciu 2001)	Distributed	N/A	N/A	E1-I3
	Becker *et al.* (2009)	Distributed	N/A	N/A	E1-I3
	Cassandra (Becker 2005)	Distributed	No	No	E1-I0
	PeerTrust (Alves *et al.* 2006)	Distributed	Distributed	No	E1-I1
		Distributed	Distributed	Distributed	E1-I2
	MTN (Zhang and Winslett 2008)	Distributed	Distributed	No	E1-I1
	GEM	Distributed	Distributed	Distributed	E1-I1

classified as E1-I3 according to the classification criteria defined in Section 3.1, that is, they do not preserve the confidentiality of neither extensional nor intensional policies. SLG identifies loops by observing goal dependencies in the "call stack" of the program; termination is detected when no more operations can be applied to the goals in the stack. SLG resolution is employed in a number of Prolog systems such as, for instance, XSB (Swift and Warren 2012). The evaluation strategy employed by GEM is similar to the XSB scheduling strategy called *local evaluation*, which completely evaluates an SCC before returning the answers of the leader to a goal outside the SCC. Similar to SLD resolution (Kowalski 1974), in TP resolution and DRA a goal is evaluated by building a single derivation tree for the goal. Loops are detected when a subgoal appears more than once in a branch of the tree, and the evaluation of a goal terminates when there are no more nodes in the derivation tree to be evaluated. OPTYap and Hulin propose a parallel tabled execution strategy to improve the efficiency of goal evaluation. OPTYap resorts to centralized data structures to identify loops and detect termination. In Hulin (1989), 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 Hu (1997) and Damásio (2000). To detect termination, the work by Hu (1997) assumes the presence of global data structures and requires goal dependencies to be propagated among the different principals. In Damásio (2000), termination detection resorts to a static dependency graph known to all principals and determined at compile time. Consequently, the confidentiality of (part of) the intensional policies is not preserved, and both algorithms are classified as E1-I2.

In trust management, distributed goal evaluation is a main issue since policies are distributed among principals. The trust management frameworks RT (Li *et al.* 2003) and Tulip (Czenko and Etalle 2007) rely on a centralized goal evaluation strategy, where all the clauses necessary for the evaluation of a goal are collected in a single location. Similarly, SecPAL (Becker *et al.* 2010) assumes all the clauses in a global policy to be available to the principal responsible for the evaluation of a goal. In SD3 (Jim and Suciu 2001), when queried for a goal, a principal returns to the requester the clauses defining the goal, with the locally defined (body) atoms already evaluated. Becker *et al.* (2009) present an algorithm in which the body atoms of the clauses defining a goal are sent in turn to the principals defining them; each principal evaluates the atom(s) defined in her policy and sends its answers and the remaining atoms to the next principal, until the evaluation fails or all atoms are evaluated. As a result, policy confidentiality is not preserved by any of these algorithms, which are thus classified as E1-I3. Furthermore, neither Jim and Suciu (2001) nor Becker *et al.* (2009) discuss how termination is detected. Cassandra (Becker 2005) employs a distributed evaluation strategy in which no information about intensional policies is disclosed. However, it does not detect neither the complete evaluation of single goals nor the termination of the whole computation.

PeerTrust (Alves *et al.* 2006) and Multiparty Trust Negotiation (MTN) (Zhang and Winslett 2008) 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 goals are quiescent. In Alves *et al.* (2006), the authors present two solutions: the first, based on the work in Damásio (2000), is also able to detect the completion of single goals, but requires the dependency graph of the global policy to be known to all principals beforehand. The second solution, which is also adopted in Zhang and Winslett (2008), detects termination of the computation without disclosing information about intensional policies. However, since all request and response messages are tagged with the identifier of the initial request, some information about goal dependencies can be inferred (hence the E1-I1 classification); more precisely, a principal can learn whether a given goal depends on a goal defined in her policy. In addition, neither PeerTrust nor MTN features a loop identification mechanism. Consequently, they are not able to detect termination of individual goals, which is required to free the resources used during the computation and to allow the use of negation. Furthermore, when using negation, the detection of loops through negation allows to preserve the soundness and completeness of the computation with respect to the standard semantics for logic programs. We enable the identification of loops and the detection of goal termination at the cost of possibly revealing more information about goal dependencies. In fact, in GEM all the principals involved in a loop are notified about the loop: on the one hand, this enables the principal(s) handling negated goals to terminate the computation with floundering. On the other hand, this implies that GEM discloses information about the presence of mutual dependencies among goals to more principals than PeerTrust and MTN. In the example in Section 4.2, for instance, with PeerTrust and MTN the research institute *ri* would not receive any loop notification from company *c*2; therefore, *ri* would not learn that there exists a mutual dependency between *memberOfAlpha(ri,X)* and *memberOfAlpha(c2,X)*.

Besides the protection of intensional policies, preserving the confidentiality of extensional policies is also an important requirement of trust management systems, as the answers of a goal might contain sensitive information (e.g., the list of patients of a mental hospital). Even though none of the existing goal evaluation algorithms satisfies this requirement (see Table 3), GEM can be easily adapted to protect the confidentiality of extensional policies. In particular, by enabling the distributed evaluation of policies, GEM allows 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 an algorithm that relies on a centralized evaluation strategy. A finer-grained protection of extensional policies can be achieved by integrating GEM with trust negotiation algorithms (Winsborough *et al.* 2000; Winslett 2003). Trust negotiation algorithms protect the disclosure of extensional policies (i.e., possibly sensitive credentials) by means of *disclosure policies* that specify which credentials a requester must provide to get access to the requested credentials. Some trust negotiation algorithms also deal with the protection of disclosure policies (e.g., Seamons *et al.* 2001); however, they assume that all the credentials of the principals in a trust management system have already been derived when a transaction takes place (Winsborough and Li 2002). GEM, on the other hand, provides a way of deriving those credentials. Thus, GEM and trust negotiation algorithms can be combined in such a way that a GEM request is evaluated only if the requester satisfies the disclosure policy of that goal, i.e., if she is trustworthy enough to see the answers to the request. The resulting integrated algorithm enables distributed goal evaluation while preserving the confidentiality of both intensional and extensional policies. A similar approach is presented in Koshutanski and Massacci (2008) and Lee *et al.* (2009). However, in Koshutanski and Massacci (2008) the authors do not discuss how to deal with recursive policy statements, while the algorithm presented in Lee *et al.* (2009) raises an error in the case that cyclic dependencies are detected, and for this reason is not complete. MTN (Zhang and Winslett 2008) also applies trust negotiation strategies to distributed goal evaluation, but as discussed in Section 7 this algorithm is not able to detect termination of individual goals within a computation. In Minami *et al.* (2011), the authors present a framework to analyze and compare distributed goal evaluation algorithms based on the information about extensional policies that they disclose during a computation.

To conclude, we point out that contrary to other works on goal evaluation (e.g., Lee *et al.* 2010)), the distributed evaluation strategy of GEM does not allow to build the complete “proof” of a goal. Building such a proof is in fact similar to constructing the derivation tree of a goal. Even though cryptographic techniques can be employed to prevent the disclosure of the facts used in the derivation process (Lee *et al.* 2009), the construction of such a proof cannot be obtained without disclosing the intensional policies of the principals involved in the evaluation, which is what GEM aims to avoid. We argue that the approach followed by GEM is consistent with the concept of trust management. In trust management, in fact, if the policy of a principal *a* refers to the policy statements of a principal *b*, then *a* trusts *b* for the definition and evaluation of those statements. When the proof of a goal is required, the confidentiality requirement should be put aside in favor of a goal evaluation strategy that allows the construction of such a proof (e.g., RT; Lee *et al.* 2003).

8 Conclusions

In this paper we have presented GEM, a distributed goal evaluation algorithm for trust management systems. Different from many of the existing algorithms, GEM detects the termination of a computation in a completely distributed way without disclosing intensional policies, thereby preserving their confidentiality. In addition, GEM is able to detect when the single goals within a computation are fully evaluated, by enabling the identification of strongly connected components. Even though this may lead to the disclosure of some additional information about goal dependencies, it also enables the use of negation (as failure) in policies. In Section 4.2 we show that the information disclosed by GEM is not sufficient to infer the intensional policy of a principal; thus, we believe that the benefits of our solution overcome the drawbacks. GEM always terminates and is sound and complete with respect to the standard semantics for logic programs. As future work, we plan to extend GEM to support constraint rules (Li and Mitchell 2003) and subsumptive tabling.

Although efficiency is not a primary objective of this paper, GEM can contribute to keep network traffic low. In fact, in most distributed goal evaluation systems (e.g., Alves *et al.* 2006) answers are sent as soon as they are computed. On the contrary, GEM delays the communication of the answers of a goal until all possible answers have been computed, i.e., until all the branches of the partial derivation tree of the goal have been inspected. This strategy may delay the identification of the answers of ground goals. However, it simplifies the termination detection mechanism and we believe reduces the number of messages exchanged by principals during a computation. The experiments presented in Section 5 suggest that since the computation time is dominated by network communication, a reduction in the number of messages exchanged between principals leads to a consistently lower computation time. In addition, since the answers of a goal can be reused for future computations, the proposed solution may reduce the computation time of later evaluations.

Based on the results of the experiments presented in Section 5, we can conclude that GEM performs well both in terms of computation time and memory occupation even for very large global policies. To confirm this conviction, we have employed GEM in some prototype of real-world distributed systems in the maritime safety and security (Trivellato *et al.* 2011) and employability (Böhm *et al.* 2010) domains. In addition, we are currently designing an advanced version of GEM that implements an "early loop detection" strategy to avoid the reevaluation of side requests. Finally, since the policy language proposed in this paper can be used to represent the semantics of several existing trust management languages (e.g., RT, Li et al. 2003; and PeerTrust, Alves *et al.* 2006), we point out that GEM can be used to evaluate goals over policies expressed in any of those languages.

Acknowledgements

This work has been carried out as part of the POSEIDON project under the responsibility of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

Alves, M., Damasio, C. V., Nejdl, W. and Olmedilla, D. 2006. A distributed tabling algorithm for rule based policy systems. In *Proc. of International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, Washington, DC, 123–132.

Apt, K. R. 1990. Logic programming. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*. MIT Press, Cambridge, MA, 493–574.

Apt, K. R., Blair, H. A. and Walker, A. 1988. *Towards a Theory of Declarative Knowledge*. Morgan Kaufmann, San Francisco, CA, 89–148.

Apt, K. R. and Bol, R. N. 1994. Logic programming and negation: A survey. *Journal of Logic Programming 19–20*, 9–71.

Apt, K. R. and Marchiori, E. 1994. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing 6,* 6A, 743–765.

Becker, M. Y. 2005. *Cassandra: Flexible Trust Management and Its Application to Electronic Health Records*. PhD thesis, Computer Laboratory, University of Cambridge, UK.

Becker, M. Y., Fournet, C. and Gordon, A. D. 2010. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security 18*, 619–665.

Becker, M. Y., Mackay, J. F. and Dillaway, B. 2009. Abductive authorization credential gathering. In *Proc. of the 10th International Conference on Policies for Distributed Systems and Networks*. IEEE Press, Piscataway, NJ, 1–8.

Blaze, M., Feigenbaum, J. and Lacy, J. 1996. Decentralized trust management. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 164–173.

Böhm, K., Etalle, S., den Hartog, J., Hütter, C., Trabelsi, S., Trivellato, D. and Zannone, N. 2010. Flexible architecture for privacy-aware trust management. *Journal of Theoretical and Applied Electronic Commerce Research 5*, 77–96.

Bradshaw, R. W., Holt, J. E. and Seamons, K. E. 2004. Concealing complex policies with hidden credentials. In *Proc. of the 11th Conference on Computer and Communications Security*, V. Atluri, B. Pfitzmann and P. D. McDaniel, Eds. ACM, New York, 146–157.

Bry, F. 1990. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *Data Knowl. Eng. 5,* 4, 289–312.

Chen, Y. 1997. Magic sets and stratified databases. *International Journal of Intelligent Systems 12,* 3, 203–231.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Costantini, S. 2001. Comparing different graph representations of logic programs under the Answer Set semantics. In *Proc. of the 1st International Workshop on Answer Set Programming*, A. Provetti and T. C. Son, Eds. AAAI Press. Available at http://www.cs.nmsu.edu/∼tson/ASP2001/7.ps

Czenko, M. and Etalle, S. 2007. Core TuLiP logic programming for trust management. In *Proc. of the 23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. Springer-Verlag, Berlin, 380–394.

Czenko, M., Tran, H., Doumen, J., Etalle, S., Hartel, P. and den Hartog, J. 2006. Nonmonotonic trust management for P2P applications. *Electronic Notes in Theoretical Computer Science 157,* 3 (May), 113–130.

Damásio, C. V. 2000. A distributed tabling system. In *Proc. of the 2nd Conference on Tabulation in Parsing and Deduction*. 65–75.

Di Marzo Serugendo, G., Foukia, N., Hassas, S., Karageorgos, A., Mostéfaoui, S. K., Rana, O. F., Ulieru, M., Valckenaers, P. and van Aart, C. 2004. Self-organisation: Paradigms and applications. *Engineering Self-Organising Systems 2977*, 1–19.

Dong, C. and Dulay, N. 2010. Shinren: Non-monotonic trust management for distributed systems. In *Proc. of the 4th International Conference on Trust Management*, M. Nishigaki, A. Jøsang, Y. Murayama and S. Marsh, Eds. IFIP, vol. 321. Springer, Boston, MA, 125–140.

Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. and Ylonen, T. 1999. SPKI Certificate Theory, RFC Editor, United States.

Fitting, M. 1985. A Kripke–Kleene semantics for general logic programs. *Journal of Logic Programming 2,* 4, 295–312.

Frikken, K., Atallah, M. and Li, J. 2006. Attribute-based access control with hidden policies and hidden credentials. *IEEE Transactions on Computers 55*, 1259–1270.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference and Symposium on Logic Programming, (ICLP'88)*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, 1070–1080.

Guo, H.-F. and Gupta, G. 2001. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proc. of the 17th International Conference on Logic Programming*, P. Codognet, Ed. Springer-Verlag, London, 181–196.

Hoch, J. and Shamir, A. 2008. On the strength of the concatenated hash combiner when all the hash functions are weak. In *Automata, Languages and Programming*, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir and I. Walukiewicz, Eds. LNCS, vol. 5126. Springer-Verlag, Berlin, 616–630.

Hu, R. 1997. *Efficient Tabled Evaluation of Normal Logic Programs in a Distributed Environment*. PhD thesis, State University of New York at Stony Brook.

Hulin, G. 1989. Parallel processing of recursive queries in distributed architectures. In *Proc. of the 15th International Conference on Very Large Data Bases*, P. M. G. Apers and G. Wiederhold, Eds. Morgan Kaufmann Publishers Inc., San Francisco, CA, 87–96.

Jim, T. and Suciu, D. 2001. Dynamically distributed query evaluation. In *Proc. of the 20th Symposium on Principles of database systems*, P. Buneman, Ed. ACM, New York, 28–39.

Koshutanski, H. and Massacci, F. 2008. Interactive access control for autonomic systems: From theory to implementation. *ACM Transactions on Autonomous and Adaptive Systems 3,* 3, 1–31.

Kowalski, R. 1974. Predicate logic as a programming language. *Information Processing 74*, 556–574.

Lee, A. J., Minami, K. and Borisov, N. 2009. Confidentiality-preserving distributed proofs of conjunctive queries. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security*, W. Li, W. Susilo, U. K. Tupakula, R. Safavi-Naini and V. Varadharajan, Eds. ACM, New York, 287–297.

Lee, A. J., Minami, K. and Winslett, M. 2010. On the consistency of distributed proofs with hidden subtrees. *ACM Transactions on Information and System Security 13,* 3, 1–32.

Leuschel, M., Martens, B. and Sagonas, K. 1998. Preserving termination of tabled logic programs while unfolding. In *Proc. of the 7th International Workshop on Logic Program Synthesis and Transformation*, N. E. Fuchs, Ed. LNCS, vol. 1463. Springer-Verlag, Berlin, 189–205.

Li, N. and Mitchell, J. C. 2003. Datalog with constraints: A foundation for trust management languages. In *Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages*, V. Dahl and P. Wadler, Eds. LNCS, vol. 2562. Springer-Verlag, London, 58–73.

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

Minami, K., Borisov, N., Winslett, M. and Lee, A. J. 2011. Confidentiality-preserving proof theories for distributed proof systems. In *Proc. of the 6th Symposium on Information, Computer and Communications Security*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu and D. S. Wong, Eds. ACM, New York, 145–154.

Park, D. 1969. Fixpoint induction and proofs of program properties. *Machine Intelligence 5,* 59–78.

Przymusinska, H. and Przymunsinski, T. C. 1990. Weakly stratified logic programs. *Fundamenta Informaticae 13,* 1, 51–65.

Przymusinski, T. C. 1988. *On the Declarative Semantics of Deductive Databases and Logic Programs*. Morgan Kaufmann, San Francisco, CA, 193–216.

Przymusinski, T. 1990. The well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae 13,* 4, 445–463.

Ramakrishnan, R. 1991. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming 11*, 189–216.

Rocha, R., Silva, F. and Costa, V. S. 2005. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming 5,* 1–2, 161–205.

Seamons, K. E., Winslett, M. and Yu, T. 2001. Limiting the disclosure of access control policies during automated trust negotiation. In *Proc. of the Network and Distributed System Security Symposium*. The Internet Society, San Diego, CA.

Shen, Y.-D., Yuan, L.-Y., You, J.-H. and Zhou, N.-F. 2001. Linear tabulated resolution based on Prolog control strategy. *Theory and Practice of Logic Programming 1*, 71–103.

Stine, K., Kissel, R., Barker, W. C., Lee, A. and Fahlsing, J. 2008. *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), Gaithersburg, MD.

Swift, T. and Warren, D. S. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1–2, 157–187.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proc. of the 3rd International Conference on Logic Programming*, E. Y. Shapiro, Ed. Springer-Verlag, London, 84–98.

Trivellato, D., Zannone, N. and Etalle, S. 2011. A security framework for systems of systems. In *Proc. of the 12th International Conference on Policies for Distributed Systems and Networks*. IEEE Computer Society, Piscataway, NJ.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38*, 619–649.

Vieille, L. 1987. A database-complete proof procedure based on SLD-resolution. In *Proc. of the 4th International Conference on Logic Programming*, J. L. Lassez, Ed. MIT Press, Cambridge, MA, 74–103.

Winsborough, W. H. and Li, N. 2002. Protecting sensitive attributes in automated trust negotiation. In *Proc. of Workshop on Privacy in the Electronic Society*, S. Jajodia and P. Samarati, Eds. ACM, New York, 41–51.

Winsborough, W. H., Seamons, K. E. and Jones, V. E. 2000. Automated trust negotiation. In *Proc. of the DARPA Information Survivability Conference and Exposition*. Vol. 1. IEEE Computer Society, Los Alamitos, CA, 88–102.

Winslett, M. 2003. An introduction to trust negotiation. In *Proc. of International Conference on Trust Management*, P. Nixon and S. Terzis, Eds. LNCS, vol. 2692. Springer-Verlag, Berlin, 275–283.

Yu, T. and Winslett, M. 2003. A unified scheme for resource protection in automated trust negotiation. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, 110–122.

Zhang, C. C. and Winslett, M. 2008. Distributed authorization by multiparty trust negotiation. In *Proc. of the 13th European Symposium on Research in Computer Security*, S. Jajodia and J. López, Eds. LNCS, vol. 5283. Springer-Verlag, Berlin, 282–299.

Zhou, N.-F. and Sato, T. 2003. Efficient fixpoint computation in linear tabling. In *Proc. of the 5th International Conference on Principles and Practice of Declarative Programming*, K. Sagonas and D. Miller, Eds. ACM, New York, 275–283.