The Efficient Maintenance of Access Roles with Role Hiding

Chaoyi Pang
eHRC
CSIRO
Australia
chaoyi.pang@csiro.au

Xiuzhen Zhang
School of CS&IT
RMIT University
Australia
xiuzhen.zhang@rmit.edu.au

Yanchun Zhang
School of CS & Maths
Victoria University
Australia
yzhang@csm.vu.edu.au

Kotagiri Ramamohanarao
Department of CSSE
University of Melbourne
Australia
rao@csse.unimelb.edu.au

Abstract

Role-based access control (RBAC) has attracted considerable research interest. However, the computational issues of RBAC models are yet to be thoroughly studied. In this paper, we study the problem of efficient maintenance of large RBAC models in a database-based multi-domain Web service environment. We propose first-order (SQL) algorithms to maintain the reachability of access roles under dynamic changes. The main advantages of our algorithms are: the support of various operations required for managing access roles with fractional information of roles; the maintenance of an update through operating a bounded number of join operations despite of the data size. To the best of our knowledge, our algorithms are the first attempt to maintain RBAC models using a first-order language.

1 Introduction

In database-based distributed Web service applications, a data service provider is the access point for data resources in the network, enabling data source specified access by users and other services. Secure and effective access control is crucial in such environments, especially when sensitive data is involved in multiple domains [19, 20]. In this situation, managing the access roles to efficiently support system-wide activities is quite challenging: it needs to support the dynamic changes on accessibility at both the service provider level and the local database level in addition to evaluating the impact of such alterations on the service.

The Role-Based Access Control (RBAC) model [13, 17, 19] has been widely accepted as an effective technique for access control. In the RBAC model, roles and their relationship are described by a *role graph* in which the nodes represent the roles in a system, and the arcs represent the accessibility between nodes, the *is-junior* relationship [13]: One role can access to another role if and only if there exists a path from the first role to the second.

In a large distributed enterprise environment, there are many *domains* that work cooperatively to provide the integrated service. As each domain is an autonomous entity that manages its own resources, protecting each domains *privacy*[1] while supporting their collaboration is highly regarded in practice. For example, suppose that there is a trusted agency that provides the information of many universities to overseas students. The agency can be seen as the service provider and the universities as domains. The agency can access, process and manipulate some of the students' data but are not allowed to store or copy the data locally for privacy considerations. Also for competitive reasons, a university may not be willing to share its data with other universities directly, but may allow its data to be used in a restricted manner: sensitive information such as names will be allowed to be seen (fully, partly or encrypted) only by certain roles in the agency rather than by roles in other universities. All these criteria suggest that the existence of a global mediated schema is obligatory and the various requirements can make the number of roles quite large and can form a complex roles hierarchical structure.

The previous example can be formulated as a RBAC model. We assume there is a service provider that is the entry point of inter-operation with the domains, and the service provider and all the domains adopt a RBAC model and are in database environments. Collaborations among domains are achieved via the service provider by invoking domain roles. The roles at the service provider carry out across-domain access controls for integrated applications. The direct communication and operations between different domains are prohibited. Models with a mediator that satisfy these design criteria have been studied in many research papers on data integration since the paper of [4].

In a large dynamic distributed enterprise environment, the service provider may receive many requests within a very short time. Obviously change requests to the RBAC model are usually much less frequent than query/access requests. The processing of authorization or rejection of a query/access request need to verify roles' reachability and can be expensive if it is done from scratch by checking the existence of a path from the user role to the domain roles.

Contributions: In our settings, the (enterprise) RBAC model is represented as a Directed Acyclic Graph (DAG).

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17–19, 2008

[1] In this paper, we use the term of privacy in its general sense.

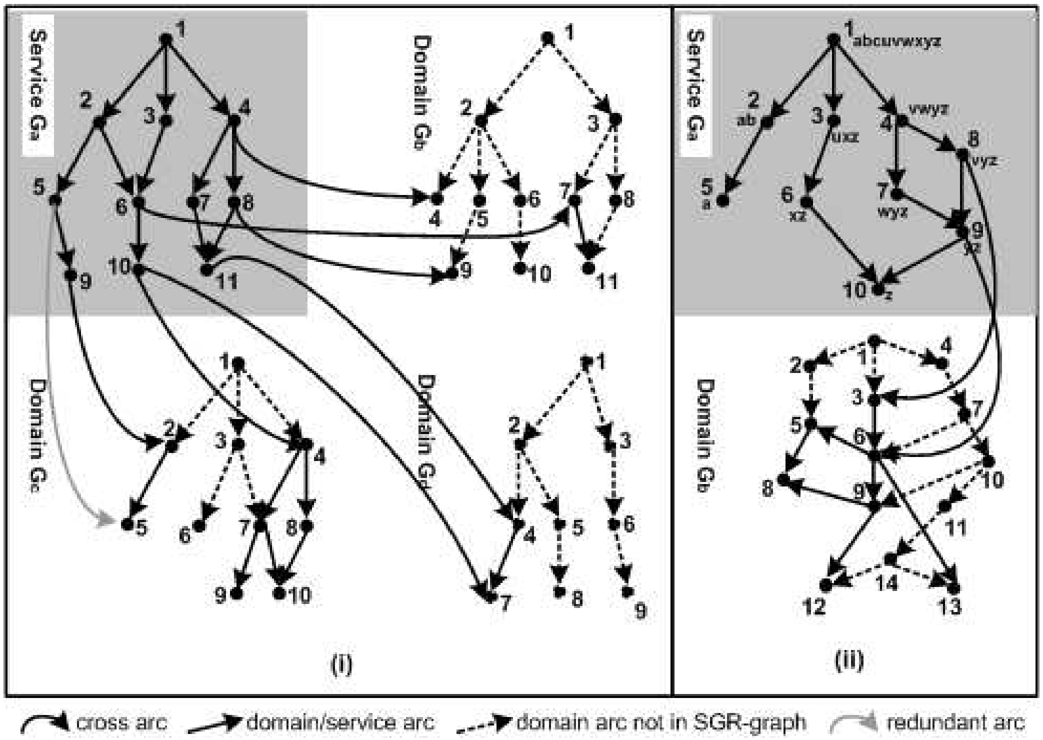

Figure 1: Domain role and global role graphs

The *transitive closure* (TC) relation on a DAG depicts roles' reachability. To alleviate this problem of accessibility, we propose to pre-compute the reachability between roles, i.e., the TC relation on role graph, to simplify the evaluation process on query requests and incrementally maintain the TC online under changes such as adding a new domain or updating an existing role. We provide first-order algorithms on the maintenance of role accessibility in the (enterprise) RBAC model that can be used for centralized models such as the models of [3, 13, 15]. Our technique implies that the maintenance can be done with a bounded number of join operations regardless of the size of a role graph. Our main contributions on the algorithms are three folds: (a) We provide two core first-order (SQL) algorithms that extend the maintenance of *TC* under a *TC-closed set* (Definition 3.1) update. Since a *TC*-closed set is not first-order expressible [2] from the sets that were previously studied on updates such as an antichain or a cartesian-closed set [7, 5], the extension is not obvious and not derivable from these sets. (b) We also show that many updates on a role graph can be decomposed into a bounded number of TC-closed sets and, therefore, can be maintained by executing the core algorithms up to a bounded number of times. The update operations that are supported in the paper are[3]: inserting/deleting a cross-domain access arc, inserting/deleting a domain, inserting/deleting a role, and inserting/deleting a privilege into/from a role. (c) Our maintenance algorithms are localized: they achieve maintenance by using partial information. This mechanism ensures better local security and privacy and allows domain roles to be hidden from the service provider. To the best of our knowledge, our maintenance algorithms, which are highly efficient, are a first attempt to use SQL for managing and maintaining role reachability in RBAC models.

In terms of maintaining RBAC models, our algorithms are distinct in several aspects. (a) Our first-order algorithms may not be applicable to the mediator-free model (SERAT) of [19] as the global role graph in a SERAT may have cycles and therefore making it hard to maintain the role accessibility with first-order queries [14]. Thus, having a central service provider mediating the services from all domains can reduce the cost for maintaining global role accessibility. (b) Since most of the conflict constraints relate to the constraints of user groups or the assignment from user groups, we only discuss the role-to-role conflicts in the paper. A role-to-role conflict means that two roles are never accessible from one to another. (c) A redundant arc in a role graph is a redundant expression on roles' subsumption and can be expressed transitively through other roles' subsumption. Redundancy in a role graph may cause difficulty in managing access roles and may make it error prone [13, 18]. Our algorithms also support the operations of redundancy detection/elimination and conflict checking.

[2] Intuitively, it means that a *TC*-closed set can not be represented by antichains and/or cartesian-closed sets under a bounded number of join operations.

[3] Since the update on an arc or a node can be implemented by first deleting the arc or node then inserting a new arc or a node, we will only study insertion and deletion operations in the paper.

In addition to redundant arcs we also introduce the notion of reducible nodes (See Section 4). Papers of [2, 10, 12] provide recursive (not first-order) algorithms for removing redundant arcs. The use of TC on checking conflict consistency and redundancy in 3-graph model has been extensively studied in [15].

The practical and theoretical significance of using a first-order language for maintaining recursive database views has been intensively addressed in [8, 9, 11, 14, 16], and the algorithms have constant parallel complexity [1, 11]. Our algorithms maintain the roles' reachability rather than the path from one role to another [19], which could be exponential to the size of the underlying graph. Our approach maintains the transitive closure of a minimal secure role graph which can be quadratic to the number of nodes in size.

2 Secure Role Graph

We describe our RBAC model in a service-based distributed environment that supports role hiding. In our model, the service provider manages the inter-operations among domains. The data of a domain cannot be fetched from other domains except from the service provider. To improve security and privacy in a collaborating environment, some roles in a domain can be hidden from the service provider. In reality, the hidden roles of a domain can be the roles the domain does not wish to be seen by the service provider. Therefore, each local domain can expose a portion of role set and their relation to the service provider, but hide the rest of the role set and hierarchical relations unknown to the service provider. We first introduce some necessary terminologies and notations used in the graph theory. We assume the reader is familiar with first-order logic or SQL. Throughout the paper, each graph is a directed acyclic graph (DAG).

A *directed graph* is a pair $G = (V, A)$, where V is a finite set of nodes and $A \subseteq V \times V$ is a set of ordered pairs or *arcs*. We will use $G(a,b)$ or $G(e)$ to denote "arc $e = (a,b)$ is in G", V(a) to denote "node a is in G". We use G_{+E} (or $G+E$ or $G \cup E$) to denote the graph resulted from inserting a set of arcs E to G, and use G_{-E} (or $G-E$) to denote the graph resulted from deleting E from G. Let $G = (V, A)$ be a graph. Suppose S and T are two subsets of A. $S \bowtie T = \{(s,t) | (\exists u) S(s,u) \wedge T(u,t) \wedge (s \neq t)\}$.

Let TC_G be the transitive closure of G, i.e., $TC_G = \{(x,y) \mid \text{there exists a path from } x \text{ to } y \text{ in } G\}$. Since arcs of the form (u,u) in a digraph contribute only in a trivial way to its transitive closure, we will only consider digraphs without such arcs. Specifically, for binary relation S, $\widehat{S} = S \cup \{(u,u) | u \text{ is a node in } S\}$. The transitive reduction [2] of DAG $G = (V, A)$ is the (unique) minimum subgraph $G_r = (V, A_r)$ of G such that $A_r \subseteq A$ and $TC_{G_r} = TC_G$. In this case, an arc of $A - A_r$ is called *redundant*. Arc $e = (u,v) \in G$ is redundant if and only if there exists a path from node u to node v in $G - e$.

In our model, the roles of domain i is modelled as DAG $G_i = (V_i, A_i)$ where the nodes set V_i represents roles and the arcs set A_i represents the dominant relationship between roles. A role is viewed as a set of privileges or permissions. An arc $(u,v) \in A_i$, which is defined on the subsumption relation on permissions, means that role u can access role v. Such role-to-role relationships are inherited transitively in the role hierarchies. We call the role graph formed by the service roles the *service graph* and denote it as G_0 or G_a. We denote domain i with G_i. A role of G_i is denoted by the *domain name* concatenating with the *role name*. For example, node 10 of domain G_b is denoted as $b10$. The inter-operation among domains is achieved by introducing *cross arcs* that make roles in domains accessible from the service provider. The set of such arcs is denoted as L. A cross arc starts from a service role and terminates at a domain role, which means that the service role can access the domain role. An non-cross arc is also called a *hierarchical arc*. The *global role graph* (GR-graph) $G_g = (V_g, A_g)$ on G_i ($i = 0, 1, ..., n$) is the graph where $V_g = \cup_{i=0}^{n} V_i$ and $A_g = (\cup_{i=0}^{n} A_i) \cup L$. Since any arc of L starts from a node of G_0 and terminates a node of $G_1 \cup G_2 \cup \ldots \cup G_n$, the GR-graph G_g is a DAG.

Symbol	Meaning
$G_i = (V_i, A_i)$	role graph at local domain i.
G_0, G_a	service (role) graph.
$G_g = (V_g, A_g)$	global role (GR) graph.
G_s, G_{ms}	a SGR graph, a minimal SGR graph.
$G_{ms}^{(n)}$	the new G_{ms} after update.
L	the set of cross arcs.
L_i	the set of cross arcs adjacent to G_i
V_{L_i}	the node set of L_i in V_i
R, R_i	a restricted-access relation on G_s, G_i
P_u,$[xy..z]$	the set of privileges of node u
Red_G	the set of redundant arcs of G
$\widehat{S}$	$S \cup \{(u,u) \| u \text{ is a node in } S\}$

Table 1: Notations

In our model, some domain conflict constraints can be automatically enforced in G_g. Let R_i be the restricted access relation on domain G_i, a subset of $V_i \times V_i$ such that $R_i \cap TC_{G_i} = \emptyset$ holds. $R_i(u,v)$ means that node u is prohibited from reaching node v in G_i. In GR-graph G_g, R_i constraints are inherently held as long as $R_i \cap TC_{G_g} = \emptyset$ holds. This suggests that, when G_i becomes G_i', $R_i \cap TC_{G_g'} = \emptyset$ holds if $R_i \cap TC_{G_i'} = \emptyset$ holds where $G_g' = (G_g - G_i) \cup G_i'$. Similar results can be derived for the restricted access $R_i'(u,v)$, which expresses that two nodes u and v are not accessible at the same time in G_i. Furthermore, in a domain, some roles and role relation (domain arcs) may not contribute to the inter-operational service: they are not accessible from the service provider G_0. For example, for reasons such as security requirements, these roles and their role-to-role relations may not be known to other domains, including the service provider. These security/privacy requirements lead to the following definition.

Definition 2.1 *Let R be a set of a restricted access rela-*

```
% INPUT: E(Start,Tail), G(Start,Tail), TC(Start,Tail).
% OUTPUT: Modified TC(Start,Tail).
%
% TABLE E(Start,Tail):
%     The set of arcs to be inserted.
% TABLE graph G(Start,Tail):
%     A role graph. For each node x, (x, x) is in G.
% TABLE TC(Start,Tail):
%     For each node pair (s,t) of TC, there is a path from s to t.
%     For each node x, (x,x) is in TC.
% TABLE Susp:
%     The suspect node pairs need to be updated when G is modified.
%
% When inserting E(Start,Tail), all paths from x through a node pair
% of E to y are affected and are stored in Susp.

    INSERT INTO Susp(Start,Tail)
    SELECT DISTINCT X.Start, Y.Tail
    FROM TC X, TC Y, E
    WHERE X.Tail=E.Start AND Y.Start=E.Tail;

% The result: Update TABLE TC

    INSERT INTO TC(Start,Tail)
    SELECT * FROM Susp;
```

Table 2: Algorithm $add(G,TC,E)$

tions. The secure global role *graph (SGR-graph)* $G_s = (V_s, A_s)$ *is a graph such that (1)* $V_s \subseteq V_g$, $L \subseteq A_s \subseteq A_g$ *and* $G_0 \subseteq G_s$*; (2)* $TC_{G_s} \cap R = \emptyset$*; (3) Every node on a path of* G_g *which starts from a node of* G_0 *going through a cross arc of* L *is in* V_s*; and (4) Every arc on a path of* G_g *which starts from a node of* G_0 *going through a cross arc of* L *is in* A_s.

Intuitively, Condition 1 means that *SGR*-graph G_s is a subgraph of G_g that contains G_0 and L; Condition 2 indicates that G_s satisfies the conflict constraints R; Condition 3 and 4 imply that each domain role or arc that can be accessed from a role of G_0 is in G_s.

Two nodes u and v of the same domain are *reducible* if u and v have different role names but have the same set of privileges. A reducible role of G_i is functionally redundant and can be produced by update operations. For instance, in Figure 1(ii), the removal of privilege u from role $a3$ makes roles $a3$ and $a6$ reducible. Unlike redundant arcs of a DAG that can be deleted entirely without affecting the graph's reachability, the reducible nodes need to be merged to preserve the graph's integrity and simplicity (Details will be discussed in Section 4). Reducible nodes lead to the following definition of minimal *SGR*-graph.

Definition 2.2 *Graph G is* minimal *if it does not have redundant arcs and reducible nodes. We denote the minimal SGR-graph* G_s *by* G_{ms}.

When a request on update occur, we maintain $TC_{G_{ms}}$ at the service provider and TC_{G_i} at domain G_i, the transitive closure of G_{ms} and G_i respectively. Maintaining $TC_{G_{ms}}$ and TC_{G_i} rather than the entire reachable relation TC_{G_g} can be more efficient than that of TC_{G_g} as the former can be smaller than the latter. It can also reduce the bottleneck caused by update operations at the service provider, as the updates on a domain will not affect other domains and the updates on the global service is minimized within G_{ms}. The major advantage of such a approach is the support of "local role hiding" and explained in the following section.

Example 2.3 *Refer to Figure 1(i). The global role graph* $G_g = G_a \cup G_b \cup G_c \cup G_d \cup L$. $L = L_b \cup L_c \cup L_d$ *is the set of cross arcs of* G_g *where* $L_b = \{(a4,b4), (a6,b7), (a8,b9)\}$, $L_c = \{(a9,c2), (a10,c4), (a5,c5)\}$ *and* $L_d = \{(a10,d7), (a11,d4)\}$. *The SGR-graph* $G_s = (V_s, A_s)$ *includes all solid arcs, which is, all arcs in the service provider* G_a*, all the cross arcs L, and the domain arcs of* $\{(b7,b11), (c2,c5), (c4,c7), (c4,c8), (c7,c9), (c7,c10), (c8,c10), (d4,d7)\}$. *The minimal SGR-graph* $G_{ms} = G_s - \{(a5,c5)\}$. *Each domain node that links with sole dotted arcs is not accessible from* G_a *and therefore, is not in either* G_s *or* G_{ms}. *Clearly,* G_{ms} *is smaller than the global role graph* G_g.

For easy reference, the symbols used throughout this paper is summarized in Table 1.

```
% INPUT: E(Start, Tail), G(Start, Tail), TC(Start, Tail).
% OUTPUT: Modified TC(Start,Tail).
%
% TABLE E(Start,Tail): The set of arcs to be deleted.
% TABLE graph G(Start,Tail):
%     A role graph. For each node x, (x,x) is also in G.
% TABLE TC(Start,Tail):
%     Each tuple (s,t) represents a path from s to t. For each node x, (x,x) is in TC.
% TABLE Susp:
%     The suspect access paths to be deleted. When deleting E(Start,Tail),
%     any path from x through a node pair of E to y are affected and stored in Susp.
    INSERT INTO Susp(Start,Tail)
    SELECT X.Start, Y.Tail
    FROM TC X, TC Y, E
    WHERE  X.Tail=E.Start AND Y.Start=E.Tail;

% TABLE Trust: the node pairs not using the deleted arcs of E.
    INSERT INTO Trust(Start,Tail)
    SELECT A.Start, A.Tail
    FROM TC A
    WHERE NOT EXISTS (SELECT * FROM Susp X
        WHERE X.Star=A.Star AND X.Tail=A.Tail);

% TABLE Temp: new node pair (u,v) represents a path from u to v.
    INSERT INTO Temp(Start,Tail)
    SELECT A.Start, B.Tail
    FROM TRUST A, G, TRUST B
    WHERE A.Tail=G.Star AND G.Tail=B.Star AND
        (NOT EXISTS (SELECT * FROM  E
            WHERE E.Star=G.Star AND E.Tail=G.Tail)) AND
        (EXISTS (SELECT * FROM Susp X
            WHERE X.Star=A.Star AND X.Tail=B.Tail));

% The result: Update TABLE TC.
    DELETE FROM TC;
    INSERT INTO TC(Start,Tail)
    (SELECT Start, Tail FROM Trust)
    UNION
    (SELECT A.Start, A.Tail FROM Temp A);
```

Table 3: Algorithm $del(G, TC, E)$

3 Updating cross-domain Arcs

To maintain $TC_{G_{ms}}$ and TC_{G_i}, we assume that the service provider stores G_0, G_{ms} and $TC_{G_{ms}}$ while each domain G_i stores G_i and TC_{G_i}. In this way, the service provider will not be able to see the full picture of its local domains G_i and each local domain G_i cannot obtain all the information of G_0, G_{ms} or $TC_{G_{ms}}$. This is a very desirable property in privacy preserving as it gives guidance to each of the participants in the system in terms of what and how they can provide their data to the services. We also assume G_i is minimal and satisfies secure requirement of $TC_{G_i} \cap R_i = \emptyset$. We will use $G_i^{(n)}$ or $G_{ms}^{(n)}$ to denote the new graphs after updating G_i or G_{ms} respectively. For each operation described in this section, a secure/conflict check should be performed after the updates. That is, if $TC_{G_i^{(n)}} \cap R_i = \emptyset$ and $TC_{G_{ms}^{(n)}} \cap R_i = \emptyset$ hold for $i = 1, \ldots, n$, then the operation is *legal*. Otherwise, the operation is *illegal*. To save space, we assume that such checks are implicitly performed routinely after each operation and we will not discuss this aspect any further in the paper.

Our technique on maintenance is based on the concept of "*TC*-closed" that will be explained in this section. Roughly speaking, our idea on maintenance is to convert an update request into the update of a bounded number of *TC*-closed sets. This guarantee that maintenance can be achieved by performing a bounded number of join operations that is irrelevant to the operated data size. For example, we will show that update a cross arc can be decomposed into two *TC*-closed sets.

In this section, we introduce the core algorithms and their applications on the maintenance of inserting/deleting a cross arc.

3.1 The Core Algorithms

The core algorithms are $add(G, TC_G, E)$ and $del(G, TC_G, D)$ where $D \subseteq A$ and $E \subset V \times V$ for $G = (V, A)$, which are listed in Table 2 and Table 3. Similar algorithms have been previously studied [5, 6, 14] for an antichain deletion[4]. The algorithms employ one common technique as the basis: A set of node pairs that depends on the deleted arcs are deleted first; this step may delete more than necessary. Then the wrong deletions are corrected through joining the result of the first step with the modified graph twice. In this section, we extend *add*() and *del*() to a *TC-closed* set insertion/deletion.

Definition 3.1 *A nonempty arc set D is called TC-closed for graph $G = (V, A)$ if (1) For any arc (u, v) of D, if there exists an arc (w, v) in $G - D$, all arcs on the path of $G \cup D$ starting from v are not in D; (2) For any two arcs (u_1, u_2) and (u_3, u_4) of D, if (u_2, u_3) is in $TC_{G_{+D}}$, each arc on the paths from u_2 to u_3 of $G \cup D$ is in D; and (3) $TC_D = D$.*

Example 3.2 *(Continued from Example 2.3). Let $D_1 = \{(c4, c7), (c4, c8), (c7, c9), (c7, c10), (c8, c10)\}$. The transitive closure of D_1 is $\vec{D}_1 = \{(c4, c7), (c4, c8), (c7, c9), (c7, c10), (c8, c10), (c4, c9), (c4, c10)\}$. $\vec{D}_1$ is TC-closed in G_{ms} and in $G_c - \{(c3, c7)\}$ but not in G_c and G_g because of the existence of arc $(c3, c7)$ that violates condition (1) of Definition 3.1. Similarly, the transitive closure $\vec{D}_2$ of $D_2 = \{(a6, b7), (b7, b11)\}$ is TC-closed in G_{ms} but not in G_g. Let $E_1 = \{(d4, d6), (d6, d9)\}$; the transitive closure $\vec{E}_1$ of E_1 is TC-closed in G_{ms} but not in G_d. From this section's result, we have $TC_{G_{ms}+E_1} = add(G_{ms}, TC_{G_{ms}}, \vec{E}_1)$ and $TC_{G_{ms}-D_i} = del(G_{ms}, TC_{G_{ms}}, \vec{D}_i)$, for $i = 1, 2$.*

Clearly, a single arc in a graph is TC-closed. We can prove that each antichain [5] is TC-closed. We also have the following property for a TC-closed set.

Property 3.3 *If D is TC-closed in G and $G' \subseteq G$, then D is TC-closed in G'.*

As the key concept in the paper, a TC-closed set is a set that can be inserted/deleted in one go during the maintenance as depicted in the core algorithms. When a request on update occur, we will convert it into a bounded number of TC-closed sets and then execute the core algorithms on each set subsequently to achieve the maintenance. It should be noted that converting an update request into antichains or cartesian-closed sets can be unbounded [5, 7].

The core algorithms are $add(G, TC_G, E)$ and $del(G, TC_G, D)$ where $D \subseteq A$ and $E \subset V \times V$ for $G = (V, A)$, which are listed in Table 2 and Table 3. Similar algorithms have been previously studied [5, 6, 14] for an antichain

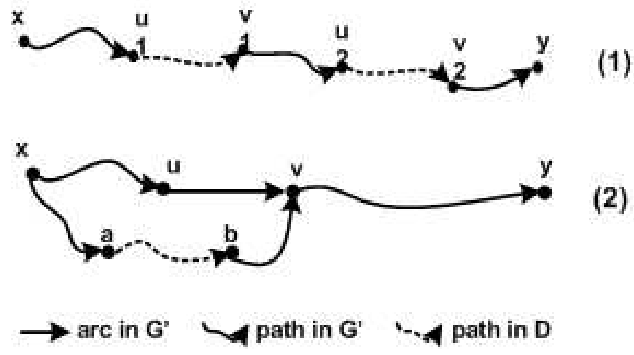

Figure 2: Proofs

deletion. The algorithms employ one common technique as the basis: A set of node pairs that depends on the deleted arcs are deleted first; this step may delete more than necessary. Then the wrong deletions are corrected through joining the result of the first step with the modified graph twice. We extend *add*() and *del*() to a *TC-closed* set insertion/deletion, as shown in Theorem 3.5.

Lemma 3.4 will be used to prove Theorem 3.5.

Lemma 3.4 *Let D be TC-closed on G. Each path of $G \cup D$ can then be formed through the concatenation of two paths in G, possibly through a path in D.*

Proof: Assuming there exists a path $path(x, y)$ in $G \cup D$, which goes through at least two paths, $path(u_1, v_1)$ and $path(u_2, v_2)$ of D as depicted in Figure 2(1) where $v_1 \neq u_2$ and $u_i \neq v_i$ for $i = 1, 2$. Since D is TC-closed on G and path $path(v_1, u_2)$ is in $G - D$, by Definition 3.1, path $path(u_2, v_2)$ is in G. Therefore, path $path(v_1, y)$ is in G. □

Theorem 3.5 *The algorithm of $add(G, TC_G, E)$ in Table 2 and the algorithm of $del(G, TC_G, D)$ in Table 3 are correct when E and D are TC-closed on DAG G.*

Proof: Since $add(G, TC_G, D)$ can be easily proven using Lemma 3.4, we only prove the result of $del(G, TC_G, D)$.

Let G' denote $G - D$. Let $\Gamma(x, y)$ express the fact that there is a walk from x to y using some node pairs $(u, v) \in D$. That is,

$$\Gamma(x, y) = \exists z_1 z_2 (TC_G(x, z_1) \wedge D(z_1, z_2) \wedge TC_G(z_2, y)).$$

Let $\Omega(x, y)$ be $G'(x, y) \vee TC_G(x, y) \wedge \neg\Gamma(x, y)$. That is, $\Omega(x, y)$ iff either (x, y) is an edge in G', or there are walks from x to y in G and no walk uses any node pairs in D.

We now show that $TC_{G'}$ can be constructed from Ω by two "joins". That is, for all (x, y), $TC_{G'}(x, y)$ iff

$$\exists uv(\Omega(x, z_1) \wedge \widehat{G}'(z_1, z_2) \wedge \Omega(z_2, y)). \tag{1}$$

Clearly, it can be seen that $\Omega(x, y)$ implies $TC_{G'}(x, y)$. So the "if" becomes obvious. For the "only if", suppose $TC_{G'}(x, y)$ holds. If this is the case, formula (1) holds for some u and v.

[4] A nonempty set of arcs D is called an *antichain* in G if, for every pair of (possibly identical) arcs (u_1, u_2) and (u_3, u_4) in D, there is no path from u_2 to u_3.

[5] A nonempty set of arcs D is called an *antichain* in G if, for every pair of (possibly identical) arcs (u_1, u_2) and (u_3, u_4) in D, there is no path from u_2 to u_3.

Assume that path $path(x,y)$ is a path in G' as depicted in Figure 2(2). Let v be the left most node on path $path(x,y)$ such that there exists a path $x...a...b...v$ where subpath $a...b$ is in D. Let arc $e=(u,v)$ be an arc of G' on path $path(x,y)$. From the choice of v, subpath $x...u$ is in $\widehat{G}'$ and $\Omega(x,u)$ holds. On the other hand, by Definition 3.1, all subpaths from v to y are in $\widehat{G}'$. Therefore $\Omega(v,y)$ holds. So the "only if" is proven. □

Let $G=(V,A)$ be the GR-graph on $G_i=(V_i,A_i)$ $(i=1,2,...,n)$ and the cross arcs set L. We will defined some SD-antichain sets used for the update. Let $TC_{\rightarrow L_i}$ be the subset of TC_G where each arc are in $G_i \cup L$ and on a shortest path terminates to a cross arc adjacent to G_i. Similarly, let $TC_{\leftarrow L_i} \subseteq TC_G$ where each each arc are in $G_i \cup L$ and on a shortest path starts from a cross arc adjacent to G_i. Clearly, $TC_{\rightarrow L_i}$ and $TC_{\leftarrow L_i} \subseteq TC_G$ are SD-antichain.

In the following, we give the general steps of computing G_{ms}, G_i, TC_{G_i} and $TC_{G_{ms}}$ when a cross arc is inserted or deleted. The update of a cross arc will not change each G_i but may alter G_{ms}.

3.2 Inserting a Cross-domain Arc

The insertion of cross arc $e=(u,v)$ does not require the subsumption $(u.P \supseteq v.P)$ on nodes u and v as a cross arc merely indicates its accessibility. Redundancy must be checked after insertions. The insertion of a cross arc may require inserting a set of domain arcs into G_{ms} in addition to the cross arc itself.

Example 3.6 *In Figure 1(i), The insertion of cross arc* $(a6,b3)$ *into* G_{ms} *requires inserting* $A_{\leftarrow a6}=\{(a6,b3),(b3,b7),(b3,b8),(b8,b11)\}$ *into* G_{ms} *and removing arc* $(a6,b7)$ *from* G_{ms} *as arc* $(a6,b7)$ *becomes redundant in the new graph.* $A_{\leftarrow a6}$ *and its transitive relation* $TC_{\leftarrow a6}$ *can be computed from* TC_{G_b}, $TC_{G_{ms}}$, G_b *and* G_{ms}. $TC_{\leftarrow a6}$ *is TC-closed in* G_{ms}. *Refer to the following general steps for details.*

The following steps are required when inserting cross arc $e=(u,v)$ where $(u\in V_0)\wedge(v\in V_i)\wedge(0<i\le n)$ holds. In this situation, G_i and TC_{G_i} are not changed $(0\le i\le n)$. Only G_{ms} and $TC_{G_{ms}}$ may be updated. If $TC_{G_{ms}}(u,v)$ holds, then e is redundant in G_{ms} and the process stops. Therefore, assume that $e\notin TC_{G_{ms}}$ holds.

1. At the Domain G_i, let $A_{\leftarrow u}$ be the set of arcs that need to be inserted. Compute $A_{\leftarrow u}$ and its transitive closure $TC_{\leftarrow u}$ through the following:
$$A_{\leftarrow u} = \{e\} \cup \{(x,y)|\widehat{TC}_{G_i}(v,x) \wedge G_i(x,y) \wedge \neg G_{ms}(x,y)\}$$
$$TC_{\leftarrow u}=\{(u,y)|\widehat{TC}_{G_i}(v,x)\wedge A_{\leftarrow v}(x,y)\} \cup\{(x,y)|\widehat{TC}_{G_i}(v,x) \wedge TC_{G_i}(x,y) \wedge \neg TC_{G_{ms}}(x,y)\}.$$

2. Send $A_{\leftarrow u}$ and $TC_{\leftarrow u}$ into the service provider G_0.

3. At the service provider G_0, do the following:

   (a) It can be proven[6] that $TC_{\leftarrow u}$ is TC-closed in G_{ms}. Therefore, $TC_{G_{ms}\cup A_{\leftarrow u}} = add(G_{ms},TC_{G_{ms}},TC_{\leftarrow u})$.

   (b) $Red_{+A_{\leftarrow u}}$, the set of redundant arcs of $G_{ms}\cup A_{\leftarrow u}$, is
$$Red_{+A_{\leftarrow u}}=\{(x,y)|G_{ms}(x,y)\wedge TC_{G_{ms}^{(n)}}(x,w_1) \wedge TC_{\leftarrow u}(w_1,w_2)\wedge TC_{G_{ms}^{(n)}}(w_2,y)\}.$$

   (c) Set $G_{ms}^{(n)}$, the new G_{ms}, to be $G_{ms}\cup A_{\leftarrow u} - Red_{+A_{\leftarrow u}}$.

For details on redundant elimination, refer to Section 4. It should be noticed that there is one call of $add()$ in the above procedure.

3.3 Deleting a Cross-domain Arc

The deletion of cross arc $e=(u,v)$ from G_{ms} means that, in addition to removing e from G_{ms}, the arcs of $G_{ms}-e$ that are not accessible from G_0 also need to be removed. Operations on redundant elimination and checking are not necessary after removing a cross arc. The maintenances of G_{ms} and $TC_{G_{ms}}$ after deleting $e=(u,v)$ involve the following steps at the service provider G_0:

1. Let $H=del(G_{ms},TC_{G_{ms}},e)$ and $G'=G_{ms}-e$.

2. Let $CK_{\leftarrow v}$ denote the set of arcs that are reached from v in G'.
$$CK_{\leftarrow v}=\{(x,y)|\widehat{H}(v,x)\wedge G'(x,y)\}.$$

3. Let $A_{\leftarrow v}$ be the set of arcs in $G_{ms}-\{e\}$ that need to be removed. For arc (x,y) of $A_{\leftarrow v}$, x is not reachable from a node of V_0 (the set of nodes of G_0). $TC_{\leftarrow v}$ is the transitive closure of $A_{\leftarrow v}$.
$$A_{\leftarrow v} = CK_{\leftarrow v}-\{(x,y)|(\exists w)CK_{\leftarrow v}(x,y) \wedge V_0(w)\wedge H(w,x)\}. \quad (2)$$
$$TC_{\leftarrow v} = A_{\leftarrow v}\cup\{(x,y)|(\exists w_1 w_2)A_{\leftarrow v}(x,w_1) \wedge A_{\leftarrow v}(w_2,y)\wedge\widehat{H}(w_1,w_2)\}. \quad (3)$$

4. Since $TC_{\leftarrow v}$ is TC-closed in G', set $G_{ms}^{(n)}$ be $G'-A_{\leftarrow v}$ and $TC_{G_{ms}^{(n)}}=del(G',H,TC_{\leftarrow v})$.

In the above procedure, $del()$ is called twice: once for deriving $A_{\leftarrow v}$ and again for obtaining $TC_{G_{ms}^{(n)}}$.

Example 3.7 *Continued from Example 3.2, the deletion of arc* $(a10,c4)$ *causes the deletion of* $A_{\leftarrow a10}=D_1\cup\{(a10,c4)\}$ *from* G_{ms}. $TC_{\leftarrow a10}$*, the transitive closure of* $A_{\leftarrow a10}$ *is TC-closed in* $G_{ms}-(a10,c4)$ *and is used to derive* $TC_{G_i^{(n)}}$ *where* $G_i^{(n)}$ *is* $(G_{ms}-A_{\leftarrow a10})$.

[6] In this situation, $TC_{\leftarrow u}$ may not be TC-closed in G_g

4 Updating Privileges on a Role

In this section, we study the maintenance of G_{ms}, $TC_{G_{ms}}$, G_i and TC_{G_i} after adding or removing a single privilege to/from a role of G_{ms} or G_i.

An access role has a set of privileges associated with it. Adding/removing a privilege to/from a role can be interpreted in two different ways: (i) Un-propagating update. That is, adding/removing a privilege to/from the role only. Other roles will not be affected; (ii) Propagating update. That is, adding a privilege to a role requires insertion of the same privilege to each of its ancestor roles; deleting a privilege from a role requires the deletion of the same privilege from each of its descendant roles.

Since the un-propagating update can be operated by first deleting the role then inserting a new role and can be easily supported with our methods. In this subsection, we will discuss the propagating update. Such an update may produce some roles that have the same set of privileges or no privileges at all. These are called reducible roles and null roles respectively. Furthermore, the update of a privilege on a role may also cause the updates on hierarchical arcs.

Example 4.1 *Refer to G_a in Figure 1(ii). In the propagating update, the deleting privilege* `z` *from node $a4$ will cause privilege* `z` *be removed from all nodes of $\{a4, a7, a8, a9, a10\}$. This results in node $a10$ becoming a null node. Similarly, the deleting privilege* `w` *from node $a4$ will cause the deletion of the same privilege from nodes $\{a4, a7\}$, resulting in node $a7$ and $a9$ being reducible. The insertion of privilege* `a` *into node $a9$ will cause the insertion of the same privilege into all nodes of $\{a4, a7, a8, a9\}$ and, subsequently, adding arc $(a9, a5)$ into G_a.*

In general, the major operations involved in the update of a privilege on a role may include: merging reducible roles; removing redundant arcs; inserting and/or deleting arcs that are induced by the merging process. The new TC is obtained from those operations and the calls of $add()$ and/or $del()$. We will study these in detail in the following. Instead of just giving the relevant algorithms, we explore the properties related to each operation to maximize the constraints on operand sets, which we believe can be beneficial for the efficiency of execution. The proof on the correctness of the procedure is omitted due to space limitation.

1. Finding the affected roles. The update(insertion or deletion) of privilege p on role u of G_i may cause other nodes of G_i to be updated. Such updates cannot be extended into another domain such as G_j for $j \neq i$. This is because that domain G_i can only be accessed from G_0 via cross arcs, where the adjacent roles of a cross arc imply that one role can access another, rather than indicate the subsumption on their privilege sets. Assume that a new privilege q is added to role $v \in G_i$ ($0 \leq i \leq n$). The privilege q should be added to each role of $\{u | TC_{G_i}(u, v)\}$ if q does not belong to it. Let α denote the set of affected nodes on which q needs to be added,

$$\alpha = \{u, v | V_i(v) \wedge TC_{G_i}(u, v) \wedge \neg P_v(q) \wedge \neg P_u(q)\}.$$

Similarly, in the case of deleting privilege p from role $u \in G_i$ ($0 \leq i \leq n$), let β denotes the set of affected nodes on which p is removed,

$$\beta = \{v, u | V_i(u) \wedge TC_{G_i}(u, v) \wedge P_u(p) \wedge P_v(p)\}.$$

With the obtained α or β, derive G_i', G_{ms}', $TC_{G_i'}$ and $TC_{G_{ms}'}$ by updating the affected nodes of G_i and G_{ms}. The process can generate reducible or null roles.

2. Processing null and reducible roles. We first show how to remove null roles from G_i', G_{ms}', $TC_{G_i'}$ and $TC_{G_{ms}'}$. We will then discuss the merge of reducible nodes.

- Removing null roles: Null roles can be induced by the removal of a privilege from a role. Since G_i is minimal, the removal of a single privilege can produce at most one null role in G_i' (It will otherwise end up with G_i having reducible roles). Let the set of arcs adjacent to the null role of G_i' be λ. Since there is no out-going arc starting from a null role,

$$\lambda = \{(u, v) | (P_v = \emptyset) \wedge (G_i'(u, v) \vee L(u, v))\}.$$

The removal of the null node v from G_i' and G_{ms}' can be achieved by deleting λ from G_i' and G_{ms}' respectively.[7] Let $G"_i = G_i' - \lambda$ and $G_{ms}" = G_{ms}' - \lambda$. It can be proven that λ is antichain. Therefore, $TC_{G"_i} = del(G_i', TC_{G_i'}, \lambda)$ and $TC_{G_{ms}"} = del(G_{ms}', TC_{G_{ms}'}, \lambda)$. To simplify notations, after removing a null node, we still use G_i' and G_{ms}' to denote $G_i"$ and $G_{ms}"$ respectively.

- Merging reducible roles: In G_i' (or G_{ms}', it can be proven that for each node, there exists at most one different node that is reducible with it. That is, reducible nodes are pairwise. Let V_n be $V_i' - V_i$, the set of (new) nodes of G_i' which are not found in G_i. Each node of V_n is updated from an affected node in G_i (in α or in β). Let V_o be $V_i \cap V_i'$, the set of (old) nodes of G_i' which are found in G_i. Similarly, $W_n = V_{ms}' - V_{ms}$ and $W_o = V_{ms} \cap V_{ms}'$. When adding a privilege into a role, the set of reducible node pairs are:

$$\begin{aligned} \mu(u, v) &= G_i'(u, v) \wedge V_n(v) \wedge V_o(u) \wedge (P_v = P_u), \\ \nu(u, v) &= G_{ms}'(u, v) \wedge W_n(v) \wedge W_o(u) \wedge (P_v = P_u). \end{aligned}$$

When removing a privilege from a role, the set of reducible node pairs are:

$$\begin{aligned} \mu(u, v) &= G_i'(u, v) \wedge V_n(u) \wedge V_o(v) \wedge (P_v = P_u), \\ \nu(u, v) &= G_{ms}'(u, v) \wedge W_n(u) \wedge W_o(v) \wedge (P_v = P_u). \end{aligned}$$

Where μ is the set of reducible node pairs of G_i' and ν is the set of reducible node pairs of G_{ms}'. We use $G_h^{(1)} = (V_h^{(1)}, A_h^{(1)})$ to denote the DAG after merging all reducible nodes of G_h' ($h = i, ms$), where $G_h^{(1)}$ and $TC_{G_h^{(1)}}$ can be obtained from:

$$V_i^{(1)} = V_i' - \{v | \mu(u, v)\}, \tag{4}$$

[7] Since a graph is expressed by its arc set relation, a node of G that is not connected to any arc of G is not in its arc set.

$$A_i^{(1)} = \{(x,y)|A_i'(x,y)\wedge\neg\mu(\_,y)\} \cup\{(x,u)|A_i'(x,v)\wedge\mu(u,v)\} \cup\{(u,y)|\mu(u,v)\wedge A_i'(v,y)\}-\mu, \quad (5)$$

$$TC_{G_i^{(1)}} = TC_{G_i^{"}} - (\{(x,v)|\mu(u,v)\wedge TC_{G_i^{"}}(x,v)\} \cup\{(v,y)|\mu(u,v)\wedge TC_{G_i^{"}}(v,y)\}), \quad (6)$$

$$V_{ms}^{(1)} = V_{ms}' - \{v|v(u,v)\},$$

$$A_{ms}^{(1)} = \{(x,y)|A_{ms}'(x,y)\wedge\neg v(\_,y)\} \cup\{(x,u)|A_{ms}'(x,v)\wedge v(u,v)\} \cup\{(u,y)|v(u,v)\wedge A_{ms}'(v,y)\}-v, \quad (7)$$

$$TC_{G_{ms}^{(1)}} = TC_{G_{ms}^{"}} - (\{(x,v)|v(u,v)\wedge TC_{G_{ms}^{"}}(x,v)\} \cup\{(v,y)|v(u,v)\wedge TC_{G_{ms}^{"}}(v,y)\}). \quad (8)$$

where $G_i^{"} = G_i' \cup \{(u,v)|\mu(v,u)\}$ and $G_{ms}^{"} = G_{ms}' \cup \{(u,v)|\ v(v,u)\}$. It can be proven that $\{(u,v)|\mu(v,u)\}$ and $\{(u,v)|v(v,u)\}$ are TC-closed in G_i' and G_{ms}' respectively. Therefore,

$$TC_{G_i^{"}} = add(G_i', TC_{G_i'}, \{(u,v)|\mu(v,u)\})$$
$$TC_{G_{ms}^{"}} = add(G_{ms}', TC_{G_{ms}'}, \{(u,v)|v(v,u)\}).$$

The next step is to remove redundant arcs from $G_i^{(1)}$ and $G_{ms}^{(1)}$.

- Eliminating redundancy: As we know, the insertion of arcs into a minimal graph may induce new redundant arcs, but it is not so for deletion. The possible steps of causing redundant arcs in $G_i^{(1)}$ and $G_{ms}^{(1)}$ are the inserted components expressed in formulae (4)-(7). Let $C_{\to U}$ and $C_{\leftarrow U}$ be components, of (4) and (5) respectively. That is,

$$C_{\to U} = \{(x,u)|A_i'(x,v)\wedge\mu(u,v)\}.$$
$$C_{\leftarrow U} = \{(u,y)|\mu(u,v)\wedge A_i'(v,y)\}.$$

We can prove that (i) each arc of $C_{\to U}$ is not redundant in $G_i^{(1)}$ and; (ii) the insertion of a non-redundant subset of $C_{\leftarrow U}$ will not induce any redundancy in $G_i^{(1)}$. These properties can be used to reduce the operation cost and improve efficiency.

Example 4.2 *In domain G_b of Figure 1(ii), merging node $b9$ to node $b6$ results in the deletion of $\{(b9,b8)$, $(b9,b12)$, $(b10,b9)\}$ and the insertion of $\{(b6,b8)$, $(b6,b12)$, $(b10,b6)\}$. Thus, the set of redundant arcs in the resulting graph is $\{(b6,b8)$, $(b7,b6)\}$. That is, the insertion of $C_{\to U} = \{(b10,b6)\}$ causes $(b7,b6)$ to become redundant; arc $(b6,b8)$ of $C_{\leftarrow U} = \{(b6,b8)$, $(b6,b12)\}$ is redundant and should not be inserted.*

It can be proven that $C_{\to U}$ and $C_{\leftarrow U}$ are antichains in $G_i^{(1)}$. The following steps are used to remove redundant arcs in $G_i^{(1)}$ through H.[8]

[8] Rather than using H, we can also use $H' = G_i' - \{(v,x)|G_i'(v,x)\wedge\mu(u,v)\}$ to compute $Red_{C_{\leftarrow U}}$.

1. Let $H = G_i^{(1)} - C_{\to U} \cup C_{\leftarrow U}$. Compute TC_H using $del()$ twice on $C_{\to U}$ and $C_{\leftarrow U}$.
2. Find redundant arcs of $C_{\leftarrow U}$ in H:
$$Red_{C_{\leftarrow U}}(u,v) = C_{\leftarrow U}(u,v)\wedge TC_H(u,v).$$
3. Insert $C = C_{\leftarrow U} - Red_{C_{\leftarrow U}}$ into H and let $H1 = H\cup C$. Compute $TC_{H1} = add(H, TC_H, C)$.
4. Insert $C_{\to U}$ into $H1$ and let $H2 = H1\cup C_{\to U}$. Here $TC_{H2} = TC_{G_i^{(1)}}$.
5. Find the redundant arcs of $H2$,
$$Red_{H2}(x,y) = H2(x,y)\wedge TC_{H2}(x,u)\wedge C_{\to U}(u,v)\wedge TC_{H2}(v,y).$$
6. Let $G_i^{(2)} = H2 - Red_{H2}$ and $TC_{G_i^{(2)}} = TC_{H2}$.
7. Similary we can derive $G_{ms}^{(2)}$ and $TC_{G_{ms}^{(2)}}$ where redundant cross arcs may exist.

3. Subsumption induced by merging roles. As shown in Example 4.1, adding a new privilege into a node can generate new subsumptions on roles and therefore, require the insertion of new arcs. In the case of adding a privilege, the newly added arcs should start at the affected nodes and terminate at unaffected nodes. These unaffected nodes have the added privilege but are not ancestors of the affected nodes. The set of new arcs to be inserted into $G_i^{(2)}$ are:

$$Ins = \{(x,y)|\alpha(x)\wedge(\neg\alpha(y))\wedge(\neg TC_{G_i^{(2)}}(x,y)) \wedge(P_y\subset P_x)\wedge(\nexists u,v)(u\neq x\vee v\neq y) \wedge(P_v\subset P_u)\wedge TC_{G_i^{(2)}}(x,u)\wedge TC_{G_i^{(2)}}(v,y)\}.$$

It can be proven that Ins is antichain. Let $G_i^{(3)}$ be $G_i^{(2)}\cup Ins$ and $TC_{G_i^{(3)}} = add(G_i^{(2)}, TC_{G_i^{(2)}}, Ins)$. The redundant arcs of $G_i^{(3)}$ are:

$$Red_{G_i^{(3)}}(x,y) = G_i^{(3)}(x,y)\wedge TC_{G_i^{(3)}}(x,u)\wedge Ins(u,v)\wedge TC_{G_i^{(3)}}(v,y).$$

We therefore have the results: $G_i^{(n)} = G_i^{(3)} - Red_{G_i^{(3)}}$ and $TC_{G_i^{(n)}} = TC_{G_i^{(3)}}$. Similarly, $G_{ms}^{(n)}$ and $TC_{G_{ms}^{(n)}}$ can be derived.

In the case of deleting a privilege, the added new arcs should start with the unaffected nodes and terminate at affected nodes. The steps of computing $G_i^{(n)}$ and $G_{ms}^{(n)}$ are omitted here as they are quite similar to the case of adding a privilege.

In the above process of inserting a privilege, $add()$ (or $del()$) can be called up to 6 times. The same measures hold for the process of deleting a privilege.

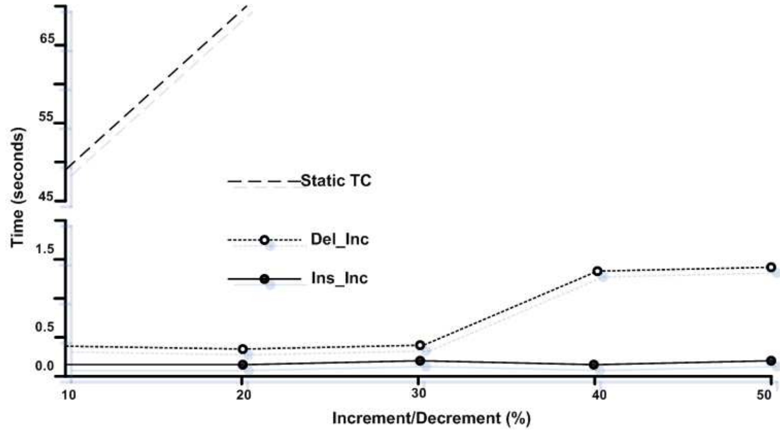

Figure 3: Runtime: The core algorithms *add*() and *del*() vs. recomputation

5 Experiments

As described in Section 3, the maintenance under different changes is based on the core algorithms *add*() and *del*(). We conducted experiments to evaluate the efficiency of these two algorithms. Our experiments were performed on a Sun SPARC machine with Oracle 9.2. Random graphs were generated to simulate the role graphs in distributed applications. The reachability among nodes (roles) for a graph was implemented as a transitive closure relation. We focused on that a set of nodes (as percentages of vertices in graphs) were added/deleted to/from random graphs. We tested the performance of the *add*() and *del*() incremental maintenance algorithms on maintaining transitive closures in comparison to recomputation.

In Figure 3 the experimental result was plotted for the algorithms on a random graph with 100 nodes, each with an average degree of 10, and 500 edges. The increment/decrement ranges from 50 edges (10%), to 250 edges (50%). Obviously, the incremental maintenance algorithms *add*() and *del*() are orders of magnitude faster than the recomputation algorithm (denoted as Static TC). In contrast to the rapid growth in computation time with respect to more edges in the graph, our incremental algorithms remain almost constant time for more edges. Particularly, in our experiments, the recomputation algorithm took from 49.73 seconds for an initial graph with 500 edges to 4:14.05 minutes for a graph with 750 edges. We have only plotted the execution time for StaticTC until the 20% increment/decrement, because its execution time for larger graphs are too large. In contrast *add*() took 0.06–0.10 seconds and *del*() took 0.34–1.49 seconds.

6 Concluding Remarks

We have studied service-based interoperation in a multi-domain environment. A centralized architecture is considered whereby a central party, called service provider, keeps information about the sharable roles of all domains and the accessibility relations on such roles. This information is stored as a transitive closure of the global role graph. Accordingly, a user request for a given service first comes to the service provider, which evaluates the authorization of this request by checking if there is connection from the user role to the domain role in the transitive closure. We propose to support role hiding, which can enforce the security and privacy of domains and can reduce the size of G_{ms}, lower the maintenance cost and diminish bottleneck at the service provider. In order to obtain the maximal number of access roles hiding from the service provider, a domain must assign the roles linked by cross arcs carefully.

We have proposed maintenance algorithms for the transitive closure of the role graphs under various update operations. The maintenance results can be extended to models where there are multiple service providers that satisfy acyclicity. The updates supported by our algorithms include the addition/deletion of a role, addition/deletion of a cross arc, addition/deletion of a privilege into/from an existing role, addition/deletion of a domain to/from the service, detection and deletion of redundant arcs, detection and merging of reducible nodes, and checking for conflicts on updates. Due to space limitation, many of them are not included in this paper.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[3] ANSI. American national standard for information technology - role based access control. In *ANSI INCITS 359-2004*, 2004.

[4] P. A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[5] G. Dong and C. Pang. Maintaining transitive closure in first-order after node-set and edge-set deletions. *Information Processing Letters*, 62(3):193–199, 1997.

[6] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/sql. *SIGMOD Record*, 29(1):44–51, 2000.

[7] G. Dong, J. Su, and R. Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):187–223, 1995. Festschrift in honor of Jack Minker.

[8] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Proc. Int'l Conference on Database Theory*, pages 282–296, Berlin, Germany, Oct. 1992.

[9] K. Etessami. Dynamic tree isomorphism via first-order updates. In *PODS*, pages 235–243, 1998.

[10] P. Gibbons, R. Karp, V. Ramachandran, D. Soroker, and R. Tarjan. Transitive compaction in parallel via branchings. *J. Algorithms*, 12(1):110–125, 1991.

[11] S. Grumbach and J. Su. First-order definability over constraint databases. In *Proceedings of Conference on Constraint Programming*, 1995.

[12] X. Han, P. Kelsen, V. Ramachandran, and R. Tarjan. Computing minimal spanning subgraphs in linear time. *SIAM J. Comput.*, 24(6):1332–1358, 1995.

[13] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.*, 2(1):3–33, 1999.

[14] C. Pang, G. Dong, and K. Ramamohanarao. Incremental maintenance of shortest distance and transitive closure in first-order logic and sql. *ACM Trans. Database Syst.*, 30(3):698–721, 2005.

[15] C. Pang, D. Hansen, and A. Maeder. Managing RBAC states with transitive relations. In *ASIACCS 2007*, 2007.

[16] S. Patnaik and N. Immerman. Dyn-FO: A parallel dynamic complexity class. In *Proc. ACM Symp. on Principles of Database Systems*, pages 210–221, 1994.

[17] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[18] B. Shafiq, J. Joshi, E. Bertino, and A. Ghafoor. Secure interoperation in a multidomain environment employing RBAC policies. *IEEE Trans. Knowl. Data Eng.*, 17(11):1557–1577, 2005.

[19] M. Shehab, E. Bertino, and A. Ghafoor. SERAT: Secure role mapping technique for decentralized secure interoperability. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 159–167, New York, NY, USA, 2005. ACM Press.

[20] R. Wonahoesodo and Z. Tari. A role based access control for web services. In *Proc. of IEEE Int'l Conference on Services Computing*, 2004.