The Specification and Enforcement of Authorization Constraints in Workflow Management Systems

ELISA BERTINO and ELENA FERRARI
Università di Milano
and
VIJAY ATLURI
Rutgers University

In recent years, workflow management systems (WFMSs) have gained popularity in both research and commercial sectors. WFMSs are used to coordinate and streamline business processes. Very large WFMSs are often used in organizations with users in the range of several thousands and process instances in the range of tens of thousands. To simplify the complexity of security administration, it is common practice in many businesses to allocate a role for each activity in the process and then assign one or more users to each role—granting an authorization to roles rather than to users. Typically, security policies are expressed as constraints (or rules) on users and roles; *separation of duties* is a well-known constraint. Unfortunately, current role-based access control models are not adequate to model such constraints. To address this issue we (1) present a language to express both static and dynamic authorization constraints as clauses in a logic program; (2) provide formal notions of constraint consistency; and (3) propose algorithms to check the consistency of constraints and assign users and roles to tasks that constitute the workflow in such a way that no constraints are violated.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General—*Security, integrity, and protection*

General Terms: Security

Additional Key Words and Phrases: Access control, authorization constraints, role and user planning

---

The work of V. Atluri was partially supported by the National Science Foundation under grant IRI-9624222, and by the National Security Agency under grant MDA904-96-1-0127.
A preliminary version of this paper, entitled "A Flexible Model Supporting the Specification and Enforcement of Role-Based Authorizations in Workflow Management Systems,"appeared in the *Proceedings of the Second ACM Workshop on Role-Based Access Control*, (Fairfax, VA., Nov. 6–7). ACM, New York, 1997, pp. 1–12.
Authors' addresses: E. Bertino and E. Ferrari, Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico 39/41, Milan, 20135, Italy; email: bertino@dsi.unimi.it; ferrarie@dsi.unimi.it; V. Atluri, Center for Information Management, Integration and Connectivity and MS/IS Department, Rutgers University, 180 University Avenue, Newark, NJ 07102; email: atluri@andromeda.rutgers.edu.

1. INTRODUCTION

Background: Workflow management systems (WFMSs) are used to run day-to-day applications in numerous application domains including office automation, finance and banking, healthcare, telecommunications, manufacturing, and production. A workflow separates the various activities of a given organizational process into a set of well-defined tasks [Georgakopoulos et al. 1995]. The various tasks in a workflow are usually carried out by several users in accordance with the organizational rules relevant to the process represented by the workflow. The stringent security requirements imposed by many workflow applications, like the ones above, call for suitable access control mechanisms. An access control mechanism enforces the security policy of the organization, typically expressed as a set of *authorizations*. This is carried out by performing a check against the set of authorizations, which are nothing but permissions granted to users, to determine whether a user wishing to execute a given action on a specified object is actually authorized to do so.

Quite often, security policies are expressed in terms of the *roles* within the organization rather than individuals (e.g., only the president can pass or veto a bill). Roles represent organizational agents to perform certain job functions within the organization. Users in turn are assigned appropriate roles based on their qualifications and responsibilities. To represent such organizational security policies directly, an access control mechanism must be capable of supporting roles. Specifying authorizations on roles is not only convenient but reduces the complexity of access control because the number of roles in an organization is significantly smaller than that of users. Moreover, the use of roles as authorization subjects, instead of users, avoids having to revoke and regrant authorizations whenever users change their positions and/or duties within the organization. Furthermore, role-based authorization is particularly beneficial in workflow environments in facilitating dynamic load balancing when a task can be performed by several individuals. As a matter of fact, commercial WFMSs, such as Lotus Notes and Action Workflow, support role-based authorizations [Georgakopoulos et al.1995; Lotus Corporation 1996; Medina-Mora et al. 1993].

Motivation: A common drawback of role-based authorization models used in current WFMSs (or even DBMSs) is their inability to model authorization constraints on roles. A typical authorization constraint, which is very relevant and well known in the security area, is *separation of duties* [Clark and Wilson 1987; Sandhu 1991]. Separation of duties aims at reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to perpetrate a fraud on his own. Separation of duties is a principle often applied in everyday life; for example, opening a safe requires two keys, held by different individuals; approval of a business trip requires permission by the department manager as well as by an accountant; and a paper submitted to a conference requires review by three different, impartial referees.

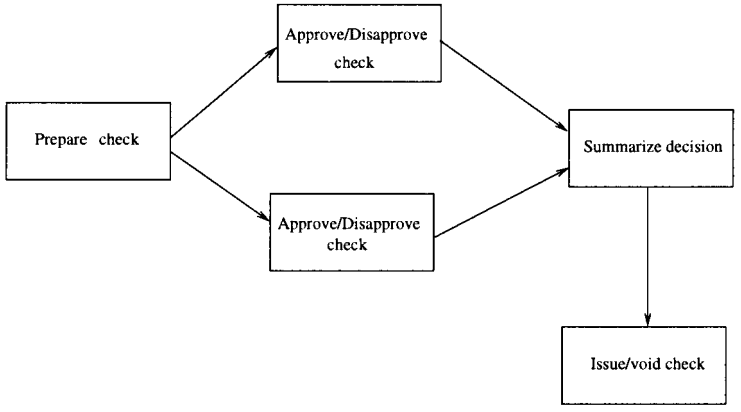

Fig. 1. Workflow representing the tax refund process.

Under the principle of separation of duties, a complex action is decomposed into several smaller steps, which are executed by different individuals. Therefore, perpetrating a fraud would require the collusion of several individuals, thus making it more difficult. Even though separation of duties has been applied to computerized information systems, current WFMSs provide no support for it. However, because a workflow decomposes a complex activity into a number of smaller well-defined tasks, we believe that separation of duties could naturally fit into workflow models.

Example 1.1 As an example, consider a simple activity dealing with a tax refund, which can be modeled by a workflow consisting of four tasks to be executed sequentially:

—Task T_1: A clerk prepares a check for a tax refund.

—Task T_2: A manager can approve or disapprove the check. This task should be performed twice by two different managers. The check will be issued if both the managers approve it; it will be voided otherwise.

—Task T_3: The decisions of the managers are collected and the final decision is made. The manager who collects the results must be different from those executing task T_2.

—Task T_4: A clerk issues or voids the check based on the result of task T_3; the clerk issuing or voiding the check must be different from the clerk who prepared the check.

The tax refund workflow is illustrated in Figure 1.

The above example illustrates the use of roles and both *static* and *dynamic* separation of duties. Each task is assigned a role; namely T_1 and T_4 must be executed by a role "clerk," whereas T_2 and T_3 must be executed by a role "manager." So the various duties, and the corresponding authorizations, are statically separated by imposing the rule that different roles

execute different tasks. An example of dynamic separation of duties is the constraint that a particular clerk must not execute both tasks T_1 and T_4 for the same check. However, he or she can perform task T_1 on some checks, while performing task T_4 for other checks. Therefore, a clerk cannot issue or void a check he or she prepared.

If there is no proper support, constraints such as separation of duties must be implemented as application code and embedded into the various tasks. Such an approach makes the specification and management of authorization constraints dificult if not impossible, given the large number of tasks that typically occur in a workflow.

Contributions: In this paper we present a language for defining constraints on *role assignment* and *user assignment* to tasks in a workflow. By using this language, we can express conditions constraining the users and roles that can execute a given task. Such constraint language supports, among other functions, both static and dynamic separation of duties. Because the number of tasks and constraints can be quite large, a relevant issue is to provide some formal notions of constraint consistency and devise algorithms for consistency checking. We show how such constraints can be formally expressed as clauses in a logic program, so that we can exploit all the results available in logic programming and deductive database areas. A further contribution of this paper is the development of algorithms for *planning* role and user assignments to the various tasks. The goal of the *planner* is to generate a set of possible assignments, so that all constraints stated as part of the authorization specification are satisfied. The planner is activated before the workflow execution starts to perform an initial plan. This plan can, however, be dynamically modified during workflow execution to account for specific situations, such as aborting a task. To our knowledge, this is the first approach proposed to systematically address the problem of assigning roles and users to tasks in a workflow.

Related work: Although the concept of roles is not new in paper-based systems, specification of access control based on roles received attention only recently [Jonscher et al. 1994; Proc. 1996; Nyanchama and Osborn 1993; Sandhu 1996; Sandhu et al. 1996]. The work by Jonscher et al. [1994] describes a role-based access control scheme for object-oriented data models. Sandhu et al. [1996] present a framework of four role-based access control models. Both papers recognize the need for constraints on roles (in fact, in many cases, constraints are the primary motivation for role-based access control). Although Sandhu et al. [1996] identify several types of constraints, Jonscher et al. [1994] model some constraints resulting from the separation of duties using the *activation conflict relation* (to reflect constraints such as that the author of a paper cannot act as its referee) and the *association conflict relation* (to capture the notion of mutually exclusive roles). However, these attempts are not sufficient to model all the constraints required in WFMSs, such as the check-preparation example outlined earlier, because they do not attempt to capture the history of events.

Significant work in this direction is due to Sandhu [1991] and Nyanchama and Osborn [1993]. To enforce separation of duties, Sandhu [1991] introduced the notion of transaction control expression, whereas Nyanchama and Osborn [1993] use history, similar to transaction control expressions. This research, however, does not focus on consistency checking. Moreover, prior research did not attempt to specify access control in terms of activities or tasks (although some preliminary effort was made in Thomas and Sandhu [1997]); so it is inadequate for WFMSs.

Other related work includes Nyanchama and Osborn's [1996], which shows how mandatory access control can be modeled in role-based security systems, and Sandhu's [1996], which shows how lattice-based mandatory access controls can be enforced using role-based access control components.

Organization of the paper: Section 2 introduces preliminary definitions and assumptions. Section 3 categorizes the various types of constraints that can be specified on workflows. Section 4 provides a logical framework to express the various types of constraints. It also presents our notion of constraint consistency. Section 5 presents a methodology to assign roles and users to tasks in a workflow, according to the specified constraints. Section 6 presents our system's architecture. Section 7 concludes the paper and outlines future work. Appendix A reports all the atoms and literals that can be expressed in our constraint specification language; formal proofs appear in Appendix B.

2. PRELIMINARIES

In our model, as in most WFMSs, we make the assumption that a workflow consists of several tasks to be executed sequentially. A task can be executed several times within the same workflow. We call such an occurrence of a given task T an *activation* of T. All activations of a task must be completed before the next task in the workflow can begin. Each task is associated with one or more roles, which are are the only ones authorized to execute a task. We allow multiple roles to be authorized for the execution of a task. We refer to the association of roles with tasks in a workflow as *workflow role specification*. In the remainder of this paper, U, $\mathcal{R}$, and $\mathcal{T}$ denote, respectively, the set of users, the set of roles, and the set of tasks in a given workflow.

A user can be authorized to *play* several roles. Moreover, a role may be played by several users. We assume that there is some mechanism that associates users with roles.[1] We assume that a user is explicitly assigned to a given role and that this assignment gives him or her the right to play the role. Whenever a user tries to execute a task, the access control system checks whether a role authorized to execute the task exists and whether

[1]A common mechanism is to maintain a table consisting of users and the list of roles the users are authorized to play. Authorizations to play a role are granted by role administrators; but these aspects are not relevant to the current paper.

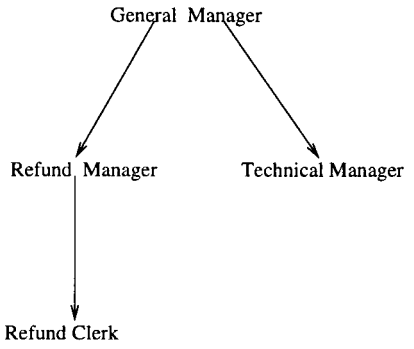

Fig. 2. An example of role order.

the user is authorized to play that role. In the following, U_i denotes the set of users authorized to play role R_i.

We assume that roles are related by a *global* partial order $\succ$. Such an order usually reflects the organizational position of roles within an organization. Let R_i, $R_j \in \mathcal{R}$ be roles. We say that R_i dominates R_j if $R_i \succ R_j$. Figure 2 illustrates an example of role order, where there is an arc from role R_i to role R_j if $R_i \succ R_j$. We assume by default that if R_i dominates R_j, then R_j should be given higher priority over R_i when assigning a role to the task. However, if no authorized user to play role R_j is available to execute the task, then the task can be executed by any user playing role R_i.

Example 2.1 Consider the roles in Figure 2 and suppose that role `Refund Clerk` is associated with task T_1 of our tax refund processing example. This means that, by default, the task of preparing a check is assigned to `Refund Clerk`. However, if no user authorized to play role `Refund Clerk` is available to execute the task, then the task can be executed by any user playing role `Refund Manager` or `General Manager`, since both `Refund Manager` and `General Manager` precede `Refund Clerk` in the role order.

Moreover, we allow the possibility of locally refining the global order. For each task of the workflow, the global order can be refined by specifying additional *local* order relationships for roles where there is no relationship in the global order. Let T_i be a task and let RS_i be the set of roles authorized to execute T_i. The partial local order for RS_i is denoted $\succ_i$. For example, suppose that $RS_i = \{\mathsf{R}_1, \mathsf{R}_2\}$ and $RS_j = \{\mathsf{R}_2, \mathsf{R}_3\}$, with $\mathsf{R}_1 \succ \mathsf{R}_2$ in the global order. The global order can be refined for task T_j by imposing $\mathsf{R}_2 \succ_j \mathsf{R}_3$ (or $\mathsf{R}_3 \succ_j \mathsf{R}_2$) because no order relationship is imposed between R_2 and R_3 in the global order. By contrast, no refinement can be performed for task T_i because the set of roles associated with task T_i is totally ordered by the global order.

As a final remark, note that if several roles are authorized to execute a task and no order for those roles is specified in the global and local orders,

the task can be performed by any of the roles. For instance, if $RS_i = \{\mathsf{R}_1, \mathsf{R}_2\}$ and none of $\mathsf{R}_1 \succ \mathsf{R}_2$, $\mathsf{R}_2 \succ \mathsf{R}_1$, $\mathsf{R}_1 \succ_i \mathsf{R}_2$, $\mathsf{R}_2 \succ_i \mathsf{R}_1$ hold, then any user who can play role R_1 or role R_2 can perform task T_i.

A workflow role specification is formally defined as follows.

Definition 2.1 (Workflow role specification). A workflow role specification W is a list of task role specifications $[\mathsf{TRS}_1, \mathsf{TRS}_2, \ldots, \mathsf{TRS}_n]$, where each TRS_i is a 3-tuple $(T_i, (RS_i, \succ_i), act_i)$ where $T_i \in \mathcal{T}$ is a task, $RS_i \subseteq \mathcal{R}$ is the set of roles authorized to execute T_i, $\succ_i$ is a local role order relationship, and $act_i \in \mathbb{N}$ is the number of possible activations of task T_i. The workflow tasks are sequentially executed in order of appearance in the workflow role specification.

In the following, we use the term workflow and workflow role specification as synonyms.

Note: from the above definition, we assume that the number of possible activations of each task within a workflow is known when the workflow role specification is given. Such a number is used by the planner when generating the actual role and user assignments for tasks. It can be considered the maximum number or the average number of expected activations. Because the planner will be re-activated during workflow execution to deal with possible execution exceptions, it is not crucial that the number of activations given by the specification be the actual one. However, a correct estimate will reduce the planning overhead at runtime.

A workflow may have several workflow instances. We assume that each instance inherits the same workflow role specification from the workflow where it was generated. We also assume that each task activation, within a workflow instance, is executed by a single user. Such a user must belong to one of the roles authorized to execute the task. However, different activations of the same task within a workflow instance may be executed by the same user or different users, depending on role authorization constraints specified for the workflow (see the next section).

Example 2.2 A possible specification for the workflow of Example 1.1 is: $W =$ [(T_1,({`Refund Clerk`}, {}), 1), (T_2,({`Refund Manager, General Manager`}, {}), 2), (T_3,({`Refund Manager, General Manager`}, {}), 1), (T_4,({`Refund Clerk,`}, {}), 1)].[2]

Throughout the paper we use W as running example. Moreover, we assume that roles in W are related according to the global role order depicted in Figure 2.

[2]'{}' denotes that no local role order is specified.

3. CONSTRAINTS ON ROLE AND USER ASSIGNMENTS

Constraints on role and user assignments to tasks in a workflow can be of several different types. In the following, we categorize them into three main categories according to the time at which they can be evaluated.

(1) *Static constraints* can be evaluated without executing the workflow.

(2) *Dynamic constraints* can be evaluated only during the execution of the workflow, because they express restrictions based on the execution history of the workflow.

(3) *Hybrid constraints*. Constraints whose satisfiability can be partially verified without executing the workflow.

Example 3.1 Consider the following constraints specified for the workflow of Example 2.2.

C_1 At least three roles must be associated with the workflow.

C_2 Task T_2 must be executed by a role dominating the roles that execute tasks T_1 and T_4, unless T_1, T_2, and T_4 are executed by the role `General Manager`.

C_3 If a user belongs to role `Refund Clerk` and has performed task T_1, then he/she cannot perform T_4.

C_4 If a user has performed task T_2, then he/she cannot perform task T_3.[3]

C_5 Each activation of task T_2 must be executed by a different user.

C_6 If more than four activations of task T_1, within the same workflow, executed by one single individual abort, then the same person cannot execute task T_1 anymore.[4]

C_7 If `Bob` executes task T_2, then he cannot execute task T_4.

Constraint C_1 is a static constraint since the number of roles associated with the workflow can be checked by simply considering the workflow role specification.

Constraints C_2, C_3, C_4, C_5, and C_7 are hybrid constraints. A preliminary consistency verification can be performed for these constraints without executing the workflow. If they are found inconsistent, they will certainly not be satisfied by the workflow execution. For instance, if a single user is associated with task T_2, constraint C_5 will never be satisfied during workflow execution. If, however, the above condition is not verified, it is necessary to check during workflow execution that whenever a user executes an activation of task T_2, the same user does not execute any further

[3]Constraints C_3 and C_4 implement dynamic separation of duties.

[4]By task abortion we mean that the task is not successfully completed, due to user errors or other errors, such as software or hardware anomalies.

Table I. Specification Predicates

Predicate	Arity	Argument types	Meaning
role	2	RT, TT	If role (R_i, T_j) is true, then role $R_i \in RS_j$;
user	2	UT, TT	If user (u_i, T_j) is true, then there exists $R_k \in RS_j$ such that $u_i \in U_k$;
belong	2	UT, RT	If belong (u_i, R_j) is true, then user $u_i \in U_j$;
glb	2	RT, TT	glb(R_i, T_j) is true if role R_i is the greatest lower bound of RS_j, wrt the global order;
lub	2	RT, TT	lub(R_i, T_j) is true if role R_i is the least upper bound of RS_j, wrt the global order;
$\succ$	2	RT, RT	$R_i \succ R_j$ is true if R_i dominates R_j in the global role order;
$\succ_k$	2	RT, RT	$R_i \succ_k R_j$ is true if R_i dominates R_j in the local role order associated with task T_k.

activation of task T_2. Similarly, constraint C_2 will never be satisfied at execution time if each role associated with task T_2 is dominated by the least upper bound of the set of roles associated with T_1 or T_4. Finally, constraint C_6 is dynamic, since no check on its consistency can be performed without executing the workflow.

4. FORMAL CONSTRAINT MODEL

In order to provide a semantic foundation for our constraint model and to formally prove consistency, we represent constraints as clauses in a normal logic program [Lloyd 1984]. We recall that clauses in a normal logic program may contain negative literals in their body.

In the remainder, we formally define the language to express authorization constraints, and then present the notion of *constraint-base*, which is the logic program encoding the constraints of a given workflow, and discuss consistency issues.

Note that the language we present is not intended as the end-user language for expressing constraints. Rather, this language is used internally by the system to analyze and enforce constraints. On top of this language, a visual programming environment can be developed along the lines of the system discussed in Chang et al. [1997].

4.1 Constraint Specification Language

We specify our constraint specification language by defining the set of constants, variables, and predicate symbols. We then define the rules that can be expressed in the language.

In the following, $\mathscr{C}$ denotes a set of constraint identifiers.

—*Constant symbols*: Every member of U (the set of users), $\mathscr{R}$ (the set of roles), $\mathscr{T}$ (the set of tasks), $\mathscr{C}$ (the set of constraints), and $\mathbb{N}$ (the set of natural numbers).

Table II. Execution Predicates

Predicate	Arity	Argument types	Meaning
$\texttt{execute}_u$	3	UT, TT, NT	If $\texttt{execute}_u(u_i, T_j, k)$ is true, then the k-th activation of task T_j is executed by user u_i;
$\texttt{execute}_r$	3	RT, TT, NT	If $\texttt{execute}_r(R_i, T_j, k)$ is true, then the k-th activation of task T_j is executed by a user belonging to role R_i;
abort	2	TT, NT	$\texttt{abort}(T_i, k)$ is true if the k-th activation of task T_i within a workflow aborts;
success	2	TT, NT	$\texttt{success}(T_i, k)$ is true if the k-th activation of task T_i within a workflow successfully executes.

—*Variable symbols*: There are five sets of variable symbols ranging over the sets U, $\mathcal{R}$, $\mathcal{T}$, $\mathcal{C}$, and $\mathbb{N}$, denoted as V_U, $V_{\mathcal{R}}$, $V_{\mathcal{T}}$, $V_{\mathcal{C}}$, and $V_{\mathbb{N}}$, respectively. In the following we denote with UT (i.e., user terms) the set $V_U \cup U$. Similarly, RT, TT, CT, NT denote the sets $V_{\mathcal{R}} \cup \mathcal{R}$, $V_{\mathcal{T}} \cup \mathcal{T}$, $V_{\mathcal{C}} \cup \mathcal{C}$, and $V_{\mathbb{N}} \cup \mathbb{N}$, respectively.

—*Predicate symbols*: The set of predicate symbols consists of five sets: (1) a set of *specification predicates* $\mathcal{SP}$, expressing information on the specification of a workflow; (2) a set of *execution predicates* $\mathcal{EP}$, capturing the effect of a workflow execution; (3) a set of *planning predicates* $\mathcal{PP}$, expressing the restrictions imposed by the constraints on the set of roles/users that can execute a task and information on constraint satisfiability; (4) a set of *comparison predicates* $\mathcal{CP}$, capturing comparison operators. $\mathcal{CP}$ includes the binary predicate $=$, whose arguments are elements in UT, RT, TT, NT, and binary predicates $<$, $\leq$, $>$, $\geq$, whose arguments are elements in NT; (5) a set of *aggregate predicates* $\mathcal{AP}$, capturing aggregate operators.[5]

Predicates belonging to $\mathcal{SP}$, $\mathcal{EP}$, $\mathcal{PP}$, and $\mathcal{AP}$ are listed in Tables I, II, III, and IV, respectively, whereas Appendix A reports all the atoms and literals that can be specified on the basis of the predicate symbols in our language.

A rule r is an expression of the form:

$$H \leftarrow A_1, \ldots, A_n, \text{ not } B_1, \ldots, \text{ not } B_m, n, m \geq 0$$

where H, $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are atoms and **not** denotes negation by failure [Lloyd 1984]. H is the *head* of the rule, whereas $A_1, \ldots, A_n$, not $B_1, \ldots,$ not B_m is the rule *body*. Rules that can be expressed in our language can be classified into a set of categories according to the predicate symbols they contain. In the following, we illustrate each of these categories.

[5]Aggregate predicates in $\mathcal{AP}$ are those defined in Das [1992].

Table III. Planning Predicates

Predicate	Arity	Argument types	Meaning
$\texttt{cannot\_do}_u$	2	UT, TT	If $\texttt{cannot\_do}_u(u_i, T_j)$ is true, then task T_j cannot be assigned to user u_i;
$\texttt{cannot\_do}_r$	2	RT, TT	If $\texttt{cannot\_do}_r(R_i, T_j)$ is true, then task T_j cannot be assigned to a user belonging to role R_i;
$\texttt{must\_execute}_u$	2	UT, TT	If $\texttt{must\_execute}_u(u_i, T_j)$ is true, then task T_j must be executed by user u_i;
$\texttt{must\_execute}_r$	2	RT, TT	If $\texttt{must\_execute}_r(R_i, T_j)$ is true, then task T_j must be executed by a user belonging to R_i;
statically_checked	1	CT	If $\texttt{statically\_checked}(C_i)$ is true, then the satisfiability of constraint C_i can be verified without executing the workflow
panic	0		If panic is true, then there exists a workflow constraint that is not satisfiable

Definition 4.1 (Explicit rule). An explicit rule is of the form $H \leftarrow$, where H is either a specification or an execution atom.

Explicit rules are facts, as their bodies are always empty. Explicit rules whose head is a specification atom express workflow role specifications, such as the roles assigned to each task and the global/local role order. These rules are automatically generated from the workflow role specification. Explicit rules whose head is an execution atom are generated either by the system during workflow execution, or by the planner to generate role (user) assignments. For example, upon the execution of the k-th activation of task T_i either the rule $\texttt{abort}\ (T_i, k) \leftarrow$, or $\texttt{success}\ (T_i, k) \leftarrow$ is generated, depending on whether the k-th activation of task T_i aborts or successfully executes.

Definition 4.2 (Assignment rule). An assignment rule takes the form $H \leftarrow L_1, \ldots, L_n$, where H is either a $\texttt{must\_execute}_u$, $\texttt{must\_execute}_r$, $\texttt{cannot\_do}_u$, or a $\texttt{cannot\_do}_r$ atom, and L_i is either a specification, execution, comparison literal, or an aggregate atom, $i = 1, \ldots, n$.

An assignment rule expresses restrictions on the set of roles/users that can execute a given task. Intuitively, assignment rules with a $\texttt{must\_execute}_r$ or $\texttt{must\_execute}_u$ as a head say that a role/user must execute a given task in order to ensure constraint consistency. Assignment rules with a $\texttt{cannot\_do}_r$ or $\texttt{cannot\_do}_u$ as a head say that a role/user must be prevented from executing a given task.

Definition 4.3 (Static checking rule). A static checking rule is of the form $\texttt{statically\_checked}(C_i) \leftarrow L_1, \ldots, L_n$, where C_i is an element of $\mathcal{C}$, and L_i is either a specification or comparison literal, or an aggregate atom,

Table IV. Aggregate Predicates

Predicate	Arity	Argument types	Meaning
count	2	conjunction of specification, execution or comparison literals, NT	count(W, n) counts number of different answers of query $\leftarrow$ W and returns this value as n;
avg	3	variable[1], conjunction of specification, execution or comparison literals, NT	avg(x, W, n) computes average values of variable x obtained from all the different answers of query $\leftarrow$ $W$and returns this value as n;
min	3	variable[1], conjunction of specification, execution or comparison literals, NT	min(x, W, n) computes minimum of the values of variable x obtained from all the different answers of query $\leftarrow$ W, and returns this value as n;
max	3	variable[1], conjunction of specification, execution or comparison literals, NT	max(x, W, n) computes maximum of the values of variable x obtained from all the different answers of query $\leftarrow$ W, and returns this value as n;
sum	3	variable[1], conjunction of specification, execution or comparison literals, NT	sum(x, W, n) computes sum of values of variable x obtained from all the different answers of query $\leftarrow$ W, and returns this value as n;

Legend:
[1]The variable is one of the bound variables of the conjunction of literals.

$i = 1, \ldots, n$. Each literal appearing in an aggregate atom of a static checking rule must be either a specification or a comparison literal.

A static checking rule states that the satisfiability of a workflow constraint can be verified without executing the workflow. These rules are automatically generated by the system to avoid redundant checks at execution time.[6]

Definition 4.4 (Integrity rule). An integrity rule is of the form panic $\leftarrow L_1, \ldots, L_n$, where L_i is a specification, execution, comparison literal, or an aggregate atom, $i = 1, \ldots, n$.

Integrity rules are used to model the nonsatisfiability of a given constraint.

Rules of our language can be further categorized into *static* and *dynamic* rules according to the time at which they can be evaluated.

Definition 4.5 (Static rule). A static rule is either an explicit, assignment, static checking or integrity rule such that all literals in the rule body are specification, aggregate, or comparison literals. Each literal in an aggregate atom of a static rule must be either a specification or a comparison literal.

[6]We will elaborate on this aspect in Section 4.3.

Table V. Constraint Specification Language Rules

Rule	Head	Body
explicit	execution or specification atom	Empty
assignment	must_execute$_u$, must_execute$_r$, cannot_do$_u$, or cannot_do$_r$ atom;	Specification, execution, or comparison literals, or aggregate atoms;
static checking	statically_checked atom;	Specification or comparison literals, or aggregate atoms. Each literal in an aggregate atom is a specification or a comparison literal;
integrity	panic atom;	Specification, execution, or comparison literals, or aggregate atoms;
static	planning or specification atom;	Specification or comparison literals, or aggregate atoms. Each literal in an aggregate atom is a specification or comparison literal;
dynamic	planning, specification , or execution atom;	Specification, execution, or comparison literals, or aggregate atoms. At least a literal in the rule must be an execution literal.

Intuitively, static rules are those that can be evaluated without executing the workflow. They are used to encode the static constraints associated with a workflow.

Definition 4.6 (Dynamic rule). A dynamic rule is an explicit, assignment, or integrity rule containing at least an execution literal. The execution literal can appear either explicitly or as an argument of an aggregate literal.

Dynamic rules must be evaluated during workflow execution. These rules are used to encode dynamic and hybrid constraints.

Table V summarizes the various types of rules supported by our language.

4.2 Constraint-Base

We associate with each workflow W a *constraint-base* (CB); that is, a set of rules specified in the constraint specification language, encoding the constraints defined on the workflow. A constraint-base is formally defined as follows.

Definition 4.7 (Constraint-base). Let W be a workflow. The constraint-base associated with W (written CB(W)), consists of a set of explicit, assignment, and integrity rules. The set of explicit rules are determined according to the conditions specified in Table VI.

We assume that each rule in the CB has a unique label assigned by the system upon its insertion. Since several rules may be necessary to encode a

single constraint, throughout this paper we use the notation $R_{i,j}$ to denote the j-th rule used to encode constraint $\mathtt{C}_i$, $i, j \in \mathbb{N}$.

Example 4.1 Consider the workflow W of Example 2.2, the global role order in Figure 2, and the constraints of Example 3.1. CB(W) is as follows:[7]

$\mathtt{R}_{1,1}$: panic $\leftarrow$ count(role (R_i, T_j), n), $n < 3$
$\mathtt{R}_{1,1}$: cannot_do$_r(R_i, \mathtt{T}_2) \leftarrow$ execute$_r(R_j, \mathtt{T}_1, k)$, $R_j \geq R_i$, $R_i \neq$ General Manager;
$\mathtt{R}_{2,2}$: cannot_do$_r(R_i, \mathtt{T}_2) \leftarrow$ execute$_r(R_j, \mathtt{T}_1, k)$, $R_j \succ R_i$, $R_i \succ R_j$;
$\mathtt{R}_{2,3}$: cannot_do$_r(R_i, \mathtt{T}_4) \leftarrow$ execute$_r(R_j, \mathtt{T}_2, k)$, $R_i \geq R_j$, $R_i \neq$ General Manager;
$\mathtt{R}_{2,4}$: cannot_do$_r(R_i, \mathtt{T}_4) \leftarrow$ execute$_r(R_j, \mathtt{T}_2, k)$, $R_j \succ R_i$, $R_i \succ R_j$;
$\mathtt{R}_{3,1}$: cannot_do$_u(U_i, \mathtt{T}_4) \leftarrow$ belong(U_i,Refund Clerk), execute$_u$ $(U_i, \mathtt{T}_1, k)$;
$\mathtt{R}_{4,1}$: cannot_do$_u(U_i, \mathtt{T}_3) \leftarrow$ execute$_u(U_i, \mathtt{T}_2, k)$;
$\mathtt{R}_{5,1}$: cannot_do$_u(U_i, \mathtt{T}_2) \leftarrow$ execute$_u(U_i, \mathtt{T}_2, k)$;
$\mathtt{R}_{6,1}$: cannot_do$_u(U_j, \mathtt{T}_1) \leftarrow$ count(abort($\mathtt{T}_1, k$), execute$_u$ $(U_j, \mathtt{T}_1, k, n)$, $n > 4$;
$\mathtt{R}_{7,1}$: cannot_do$_u$(Bob, $\mathtt{T}_4$) $\leftarrow$ execute$_u$(Bob, $\mathtt{T}_2$, k).

Moreover, let {John, Mary, Tom} be the users authorized to play the role Refund Manager, {Ken, Meg} be the users authorized to play the role General Manager, and {Bob, Sam, Matt, Alice} be the users authorized to play the role Refund Clerk. According to the rules in Table VI, CB(W) contains the explicit rules listed in Figure 3.

Note that the rule generation process is driven by the task execution order. The task execution order is given by the order in which the tasks appear in the workflow role specification (cfr., Definition 2.1). For instance, consider constraint $\mathtt{C}_3$ of Example 3.1. Such constraint is encoded by rule $\mathtt{R}_{3,1}$ of Example 4.1 because we know that task $\mathtt{T}_1$ is always executed before $\mathtt{T}_4$. If no information on the task order is known, the rule cannot_do$_u(U_i$, $\mathtt{T}_1) \leftarrow$ belong(U_i, Refund Clerk), execute$_u(U_j, \mathtt{T}_4, k)$ would also be generated.

In order to completely define the notion of CB, a semantics must be assigned to it. We consider *stable model semantics* of logic programs with negation [Gelfond and Lifschitz 1988] to identify the models of a CB. The following proposition ensures the uniqueness of the CB model.

PROPOSITION 4.1 *Any CB is a stratified normal program. Hence, it has a unique stable model.*

[7]For brevity, in the example we use the form $R_j \geq R_i$ as a shortcut for the disjunction (using two clauses) $R_j \succ R_i$ and $R_j = R_i$.

Table VI. CB Explicit Rules

Rule	Condition
$R_i \succ R_j \leftarrow$	$\forall R_i, R_j \in \mathcal{R}$ such that R_i dominates R_j in the global role order;
$R_i \succ_k R_j \leftarrow$	$\forall T_k \in \mathcal{T}$, $\forall R_i, R_j \in RS_k$ such that R_i dominates R_j in the local role order associated with task T_k;
$\texttt{role}(R_i, T_j) \leftarrow$	$\forall R_i \in \mathcal{R}$, $\forall T_j \in \mathcal{T}$, such that $R_i \in RS_j$;
$\texttt{user}(u_i, T_j) \leftarrow$	$\forall u_i \in U$, $\forall T_j \in \mathcal{T}$ such that $\exists R_k \in RS_j$ with $u_i \in U_k$;
$\texttt{belong}(u_i, R_j) \leftarrow$	$\forall u_i \in U$, $\forall R_j \in \mathcal{R}$ such that $u_i \in U_j$;
$\texttt{glb}(R_j, T_i) \leftarrow$	$\forall R_j \in \mathcal{R}$, $\forall T_i \in \mathcal{T}$ such that $R_j = glb(RS_i)$ wrt the global order;
$\texttt{lub}(R_j, T_i) \leftarrow$	$\forall R_j \in \mathcal{R}$, $\forall T_i \in \mathcal{T}$ such that $R_j = lub(RS_i)$ wrt the global order;
$\texttt{execute}_r(R_i, T_j, k) \leftarrow$	$\forall R_i \in \mathcal{R}$, $\forall T_j \in \mathcal{T}$, $\forall k \in \mathbb{N}$ such that R_i executes the k-th activation of T_j;
$\texttt{execute}_u(u_i, T_j, k) \leftarrow$	$\forall u_i \in U$, $\forall T_j \in \mathcal{T}$, $\forall k \in \mathbb{N}$ such that user u_i executes the k-th activation of T_j;
$\texttt{abort}(T_i, k) \leftarrow$	$\forall T_i \in \mathcal{T}$, $\forall k \in \mathbb{N}$ such that the k-th activation of T_i aborts;
$\texttt{success}(T_i, k) \leftarrow$	$\forall T_i \in \mathcal{T}$, $\forall k \in \mathbb{N}$ such that the k-th activation of T_i successfully executes.

General Manager $\succ$ Refund Manager $\leftarrow$	belong(U_i,Refund Manager) $\leftarrow$, $\forall U_i \in$ {John, Mary, Tom}
General Manager $\succ$ Refund Clerk $\leftarrow$	belong(U_i,General Manager) $\leftarrow$, $\forall U_i \in$ {Ken, Meg}
Refund Manager $\succ$ Refund Clerk $\leftarrow$	belong(U_i,Refund Clerk) $\leftarrow$, $\forall U_i \in$ {Bob, Sam, Matt, Alice}
role(Refund Clerk,T_j) $\leftarrow$, $j = 1, 4$	glb(General Manager, T_i) $\leftarrow$, $i = 1, \ldots, 4$
lub(Refund Clerk, T_i) $\leftarrow$, $i = 1, 4$	role(R_i, T_j) $\leftarrow$, $\forall R_i \in$ {Refund Manager, General Manager}, $j = 2, 3$
lub(Refund Manager,T_i) $\leftarrow$, $i = 2, 3$	user(U_i, T_j) $\leftarrow$, $\forall U_i \in$ {Bob, Sam, Matt, Alice}, $j = 1, 4$
	user(U_i, T_j) $\leftarrow$, $\forall U_i \in$ {John, Mary, Tom, Ken, Meg}, $j = 2, 3$

Fig. 3. Explicit rules for the workflow of Example 2.2.

We refer the reader to Appendix B for the formal proof.

Proposition 4.1 ensures that the stable model of a CB is identical to the *well-founded* model [Van Gelder et al. 1991]. In the following, we use M(CB) to denote the meaning of a CB with respect to stable model semantics.

4.3 Enhancing the CB for Consistency Checking

In the previous section we showed how workflow constraints can be naturally encoded by a logic program (i.e., the CB) expressed in our constraint specification language. The next step is to verify the consistency of the constraints with respect to the workflow specification. (Consistency checking is dealt with in the following section.) Here we explain how the CB can be modified to make the consistency checking process more efficient.

The CB of a workflow encodes static constraints by a set of static rules, and hybrid and dynamic constraints by a set of dynamic rules.[8] This implies that, at static time, we are only able to verify the consistency of static constraints.

However, for some hybrid constraints, it is possible to perform a preliminary static analysis on their consistency with respect to workflow specification.

The following example elucidates the discussion.

Example 4.2 It is possible to statically verify that constraint $\mathtt{C}_5$ of Example 3.1 will never be satisfied if the number of users authorized to execute task $\mathtt{T}_2$ is less than 2. Similarly, if Bob is not among the users authorized to execute both $\mathtt{T}_2$ and $\mathtt{T}_4$, no further check on $\mathtt{C}_7$ at execution time is needed, since no execution can violate $\mathtt{C}_7$. Thus constraint $\mathtt{C}_7$ can be pruned from the CB before workflow execution.

Note that static conditions for checking the consistency of hybrid constraints are crucial in reducing the overhead of constraint checking at execution time. Therefore, before executing consistency checking for the CB, we complement it with a set of *static consistency rules*. Static consistency rules are static rules, expressed in our constraint specification language, encoding the checks that can be performed on the consistency of hybrid constraints without executing the workflow. These rules are automatically generated by the system on the basis of the rules in the CB encoding hybrid constraints.

Example 4.3 Consider the CB of Example 4.1. The following static consistency rules are generated by the system:

$$\mathtt{R}_{2,5} : \mathtt{cannot\_do}_r(R_i, \mathtt{T}_2) \leftarrow \mathtt{role}(R_i, \mathtt{T}_2), \mathtt{lub}(R_j, \mathtt{T}_1), R_j \succ R_i;$$
$$\mathtt{R}_{2,6} : \mathtt{cannot\_do}_r(R_i, \mathtt{T}_1) \leftarrow \mathtt{role}(R_i, \mathtt{T}_1), \mathtt{glb}(R_j, \mathtt{T}_2), R_i \succ R_j;$$
$$\mathtt{R}_{2,7} : \mathtt{cannot\_do}_r(R_i, \mathtt{T}_2) \leftarrow \mathtt{role}(R_i, \mathtt{T}_2), \mathtt{lub}(R_j, \mathtt{T}_4), R_j \succ R_i;$$
$$\mathtt{R}_{2,8} : \mathtt{cannot\_do}_r(R_i, \mathtt{T}_4) \leftarrow \mathtt{role}(R_i, \mathtt{T}_4), \mathtt{glb}(R_j, \mathtt{T}_2), R_i \succ R_j;$$
$$\mathtt{R}_{5,2} : \mathtt{panic} \leftarrow \mathtt{count}(\mathtt{user}(U_i, \mathtt{T}_2), n\ ), n < 2;$$
$$\mathtt{R}_{7,2} : \mathtt{statically\_checked}(\mathtt{C}_7) \leftarrow \mathtt{not}\ \mathtt{user}(\mathtt{Bob}, \mathtt{T}_2);$$
$$\mathtt{R}_{7,3} : \mathtt{statically\_checked}(\mathtt{C}_7) \leftarrow \mathtt{not}\ \mathtt{user}(\mathtt{Bob}, \mathtt{T}_4).$$

For instance, consider the static consistency rules generated for constraint $\mathtt{C}_2$ of Example 3.1. Constraint $\mathtt{C}_2$ requires that task $\mathtt{T}_2$ must be executed by a role dominating the roles that execute tasks $\mathtt{T}_1$ and $\mathtt{T}_4$, unless $\mathtt{T}_1$, $\mathtt{T}_2$, and $\mathtt{T}_4$ are executed by the role General Manager. This means that if a role R_i is dominated by the least upper bound of the roles associated with task $\mathtt{T}_1$, then it cannot execute task $\mathtt{T}_2$, because the execution would violate constraint $\mathtt{C}_2$. Similarly, a role cannot execute task $\mathtt{T}_1$ if it domi-

[8] Recall that static rules can be evaluated before executing the workflow, whereas evaluation of dynamic rules requires the execution of the workflow.

nates the greatest upper bound of the roles associated with task T_2. Note that these checks can be done statically, that is, without executing the workflow, since they require the analysis of the workflow role specification only. These checks are encoded by rules $R_{2,5}$ and $R_{2,6}$ above. Rules $R_{2,7}$ and $R_{2,8}$ encode similar checks for task T_4.

The result in Proposition 4.1 naturally extends to CBs also containing static consistency rules.

In the following, by CB we refer to a CB complemented with the corresponding static consistency rules.

4.4 CB Consistency

Intuitively, a CB is *consistent* if and only if the constraints it encodes are satisfiable. The consistency of a CB is determined by computing and analyzing its model. The formal definition of a consistency notion for a CB entails that a number of different conditions be satisfied by the CB model. First, if the predicate `panic` belongs to the CB model, it means that at least one constraint cannot be satisfied. Therefore, a first consistency condition requires for predicate `panic` not to belong to the CB model. Then, it is necessary to verify that the facts belonging to the CB model do not express contradictory information. As an example, suppose that both $\texttt{must\_execute}_u(\texttt{John}, T_1)$ and $\texttt{cannot\_do}_u(\texttt{John}, T_1)$ belong to the CB model. This means that the workflow constraints are inconsistent since no execution of task T_1 will satisfy the constraints. Similar considerations apply to $\texttt{must\_execute}_r$ and $\texttt{cannot\_do}_r$ predicates.

To check that no contradictory information is entailed by a CB, we compute for each task T_i in a workflow W the set of roles/users that must be prevented/obliged to execute task T_i, according to the rules in CB (W). These sets are as follows:

—*Denied_Roles*$(T_i) = \cup \{R_j \mid \texttt{cannot\_do}_r(R_j, T_i) \in M(\text{CB}(W))\}$.
Denied_Roles(T_i) represents the set of roles that cannot execute task T_i according to the rules in CB(W);

—*Obliged_Roles*$(T_i) = \cup \{R_j \mid \texttt{must\_execute}_r(R_j, T_i) \in M(\text{CB}(W))\}$.
Obliged_Roles(T_i) denotes the set of roles that are obliged to execute task T_i according to the rules in CB(W);

—*Denied_Users*$(T_i) = \cup \{u_j \mid \texttt{cannot\_do}_u(u_j, T_i) \in M(\text{CB}(W))\}$.
Denied_Users(T_i) denotes the set of users that cannot execute task T_i according to the rules in CB(W);

—*Obliged_Users*$(T_i) = \cup \{u_j \mid \texttt{must\_execute}_u(u_j, T_i) \in M(\text{CB}(W))\}$.
Obliged_Users(T_i) denotes the set of users that are obliged to execute task T_i according to the rules in CB(W);

It is then necessary to verify that the roles (users) for which it is mandatory to execute a task, if any, belong to the set of roles (users) assigned to the task and are not included in the set of roles (users) which must not execute the task. Finally, it is necessary to verify that for each task there exists at least a user playing a certain role that, when assigned to the task, ensures constraint satisfiability.

The above requirements are formalized by the following definition.

Definition 4.8 (Constraint-base consistency). Let CB(W) be the constraint-base of a workflow W. CB(W) is consistent if and only if the following conditions hold:

—panic $\notin M(\mathrm{CB}(W))$;

—$\forall T_i$ of W:
  —If $Obliged\_Users(T_i) \neq \emptyset$, then:
    $Obliged\_Users(T_i) \cap Denied\_Users(T_i) = \emptyset$;
    $Obliged\_Users(T_i) \subseteq \{u \mid u \in U_j \wedge R_j \in RS_i\}$;
  —If $Obliged\_Roles(T_i) \neq \emptyset$, then:
    $Obliged\_Roles(T_i) \cap Denied\_Roles(T_i) = \emptyset$;
    $Obliged\_Roles(T_i) \subseteq RS_i$;
    $Obliged\_Users(T_i) \subseteq \{u \mid u \in U_j \wedge R_j \in Obliged\_Roles(T_i)\}$;
  —If $Obliged\_Users(T_i) = \emptyset$ and $Obliged\_Roles(T_i) = \emptyset$, then:
    $\exists R_j \in \{RS_i \setminus Denied\_Roles\ (T_i)\}$ such that: $U_j \setminus Denied\_Users(T_i) \neq \emptyset$.

5. CONSISTENCY ANALYSIS AND PLANNING

In this section we present our methodology for assigning roles and users to the tasks of a workflow according to the constraints encoded in the CB. The various steps of our methodology are illustrated in Figure 4. Each step is performed by a specific component of the authorization system. A first step, referred to as *static analysis*, determines whether the static part of the CB, that is, the subset of the CB containing only static rules, is consistent according to Definition 4.8. If the check fails, the constraints specified for the workflow are inherently inconsistent, and therefore no assignment to tasks is generated. Thus, the system security officer (or DBA or workflow designer) has to modify role assignments to tasks and/or the constraints. If the check succeeds, the *pruning* phase is executed. This phase modifies the workflow role specification to take into account the results of the static analysis phase. Moreover, CB is modified to eliminate redundant rules. Pruning will make the execution of the subsequent phases more efficient. The *planning* phase receives the modified workflow and the modified CB generated by the pruning phase as input and generates role/user assignments to tasks that satisfy the constraints, that is, roles/users, when assigned to tasks, make the CB consistent. If no assignment can be generated, an error is returned to the system security officer. To limit the

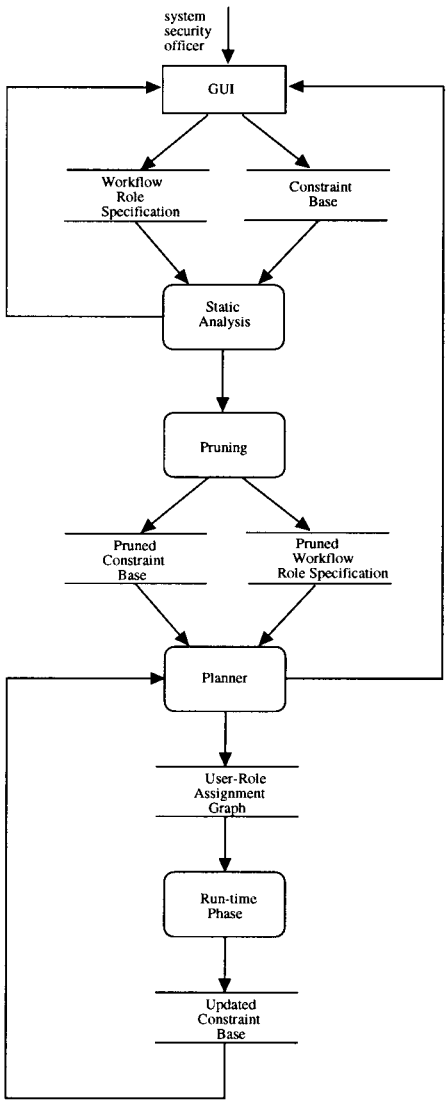

Fig. 4. Phases in constraint specification analysis and enforcement.

number of choices to be evaluated by the planner, we make the assumption that all the tasks execute successfully. At runtime, when a task aborts, constraints involving abort predicates are analyzed to verify whether the planner must be invoked again. The planner is also re-activated if the number of task activations exceed the number stated in the workflow role specification. In the remainder of this section, we discuss each phase of our methodology in detail.

Algorithm 5.1 *The static analysis algorithm*

INPUT: 1) A workflow $W = [\mathtt{TRS}_1, \mathtt{TRS}_2, \ldots, \mathtt{TRS}_n]$, $\mathtt{TRS}_i = (T_i, (RS_i, \succ_i), act_i)$, $i = 1, \ldots, n$
2) CB(W)

OUTPUT: 1) FALSE if the static part of CB(W) is inconsistent, otherwise:
2) The sets *Denied_Roles*(T_i), *Obliged_Roles*(T_i), *Denied_Users*(T_i), *Obliged_Users*(T_i), $\forall T_i$ in $\mathtt{TRS}_i$, $\forall i = 1, \ldots, n$
3) The model of the static part of CB(W)

1. Let SCB(W) be the subset of CB(W) containing only static rules
2. Compute the model of SCB(W)
3. **For** each $\mathtt{TRS}_i = (T_i, (RS_i, \succ_i), act_i) \in W$:
   Compute *Obliged_Roles*(T_i), *Denied_Roles*(T_i), *Obliged_Users*(T_i) and *Denied_Users*(T_i), wrt M(SCB(W))
   endfor
4. **If** SCB(W) is inconsistent: return FALSE

Fig. 5. The static analysis algorithm.

5.1 Static Analysis Phase

The static analysis phase verifies the consistency of the static part of the CB. Figure 5 shows an algorithm implementing the static analysis phase. The algorithm receives as input a workflow W and the associated constraint-base CB(W) and checks the consistency of the static part of CB(W), according to the conditions stated in Definition 4.8. If the check fails, FALSE is returned; otherwise the algorithm returns the model of the static part of CB(W) and the sets *Denied_Roles*(T_i), *Obliged_Roles*(T_i), *Denied_Users*(T_i), and *Obliged_Users*(T_i), for each T_i in the input workflow.

Example 5.1 Consider the CB of Example 4.1, complemented by the static consistency rules of Example 4.3. The static part of CB(W) is composed by rules $\mathsf{R}_{1,1}$, $\mathsf{R}_{2,5}$, $\mathsf{R}_{2,6}$, $\mathsf{R}_{2,7}$, $\mathsf{R}_{2,8}$, $\mathsf{R}_{5,2}$, $\mathsf{R}_{7,2}$, $\mathsf{R}_{7,3}$, and by all the explicit rules in Figure 3. Thus the model of the static part of CB(W) consists of facts shown in Figure 6. So *Obliged_Roles*(T_i) = *Denied_Roles*(T_i) = *Obliged_Users*(T_i) = *Denied_Users*(T_i) = $\emptyset$, $i = 1, \ldots, 4$. Thus, the consistency of the static part of CB(W) can be verified immediately.

5.2 Pruning Phase

The aim of the pruning phase is to modify the workflow specification according to the result of the static analysis phase, in order to efficiently execute the subsequent phases. The static analysis phase determines a set of roles for each task, if any, for which it is mandatory to execute the task (i.e., the set *Obliged_Roles*), and a set of *incorrect* role assignments, that is, a set of roles that, when assigned to a task, do not satisfy the constraints (i.e., the set *Denied_Roles*). If *Obliged_Roles* is not empty, all the roles that do not belong to this set can be removed from the specification because

General Manager $\succ$ Refund Manager	General Manager $\succ$ Refund Clerk
Refund Manager $\succ$ Refund Clerk	role(Refund Clerk,T_j), $j = 1, 4$
role(R_i, T_j), $\forall R_i \in$ {Refund Manager,General Manager}, $j = 2, 3$	
user(U_i, T_j), $\forall U_i \in$ {Bob, Sam, Matt, Alice}, $j = 1, 4$	
user(U_i, T_j), $\forall U_i \in$ {John, Mary, Tom, Ken, Meg}, $j = 2, 3$	
belong(U_i,Refund Manager), $\forall U_i \in$ {John, Mary, Tom}	
belong(U_i,General Manager), $\forall U_i \in$ {Ken, Meg}	
belong(U_i,Refund Clerk), $\forall U_i \in$ {Bob, Sam, Matt, Alice}	
glb(General Manager, T_i), $i = 1, \ldots, 4$	lub(Refund Clerk, T_i), $i = 1, 4$
statically_checked(C_7)	lub(Refund Manager,T_i), $i = 2, 3$

Fig. 6. Model of the static part of CB for the workflow in Example 5.1.

their assignment to the task violates workflow constraints. If set *Obliged_Roles* is empty, then the roles belonging to *Denied_Roles* are *pruned* from the set of roles that can be assigned to the task.

Finally, based on the results of the static analysis phase, the pruning phase updates the CB by removing the dynamic rules whose evaluation is no longer necessary in the subsequent phases. These rules are the ones that encode a constraint C_i such that statically_checked(C_i) belongs to the model of the static part of the CB.

An algorithm implementing the pruning phase is reported in Figure 7. The algorithm receives as input the results of the execution of Algorithm 5.1 on W and CB(W) and returns the pruned workflow role specification and the pruned CB(W), computed according to the methods sketched above.

Example 5.2 Consider the workflow of Example 2.2. $Prun\_W = W$, whereas $Prun\_CB = CB \setminus \{R_7\}$, since statically_checked(C_7) $\in M(SCB(W))$.

5.3 Planning Phase

This phase generates the set of possible assignments of roles and users to tasks so that all the constraints associated with the workflow are satisfied. The planning phase, intended to perform an initial plan, is executed before the workflow execution starts.

We begin with the role assignment to tasks because, when compared to the number of users, fewer roles are typically assigned for each task. Because the roles are not many, performing a correct assignment is crucial. Once the role assignment is planned, we assign users to each task. Therefore the planning phase consists of two subphases: the *role planning phase* and the *user planning* phase.

5.3.1 *Role Planning Phase.* Our approach to planning is based on the use of the CB as an hypothetical reasoner. For example, to determine whether the execution of the k-th activation of task T_j by a role R_i violates the constraints, we insert the rules, called *assignment hypotheses*, execute$_r$(R_i, T_j, k) $\leftarrow$ and success(T_j, k) $\leftarrow$ into CB. We then formulate assignment hypotheses for all the other tasks in the workflow and verify

Algorithm 5.2 *The pruning algorithm*

INPUT: 1) The output of Algorithm 5.1 when its input are W and CB(W)
OUTPUT: 1) The pruned workflow $Prun\_W$ obtained by substituting each $\mathtt{TRS}_i = (T_i, (RS_i, \succ_i), act_i)$ in W with $\mathtt{TRS}'_i = (T_i, (Upd\_RS_i, \succ_i), act_i)$
2) The pruned Constraint-Base $Prun\_CB(W)$

1. **For** each $\mathtt{TRS}_i = (T_i, (RS_i, \succ_i), act_i) \in W$:
   a. $Upd\_RS_i := RS_i$
   b. **If** $Obliged\_Roles(T_i) \neq \emptyset$: $Upd\_RS_i := Obliged\_Roles(T_i)$
   c. **If** $Obliged\_Roles(T_i) = \emptyset$: $Upd\_RS_i := RS_i \setminus Denied\_Roles(T_i)$
   d. Add $(T_i, (Upd\_RS_i, \succ_i), act_i)$ to $Prun\_W$
   endfor
2. $Prun\_CB(W) := CB(W)$
3. **For** each constraint C_i associated with W:
   If `statically_checked`$(C_i) \in M(SCB(W))$:
   Let DCB(W) be the subset of CB(W) consisting only of dynamic rules
   Let $S = \{R_{i,j} \mid R_{i,j} \in DCB(W)\}$
   Remove S from $Prun\_CB(W)$
   endif
   endfor

Fig. 7. The pruning algorithm.

the consistency of CB. We then remove the hypotheses $\mathtt{execute}_r(R_i, T_j, k) \leftarrow$ and $\mathtt{success}(T_j, k) \leftarrow$ from the CB and consider another role for T_j. Thus planning generates all consistent assignment hypotheses.

To keep role planning feasible, we make the assumption that all the activations of a task must be executed by the same role. In the algorithm, we use a slightly different notion of consistency than the one in Section 4.4, in that we require, in addition to the conditions in Definition 4.8, that for each task T_i in the workflow, $Obliged\_Roles(T_i)$ contains at most one element. In planning the role assignments, we suppose that each task T_i is executed at most act_i times, where act_i is the number of activations specified in the workflow role specification.

Role assignments are represented by a *role assignment graph*, defined as follows.

Definition 5.1 (Role assignment graph). Let $W = [\mathsf{TRS}_1, \mathsf{TRS}_2, \ldots, \mathsf{TRS}_n]$ be a workflow role specification with $\mathsf{TRS}_i = (T_i, (RS_i, \succ_i), act_i)$, $i = 1, \ldots, n$. The role assignment graph of $W(RAG(W))$ is a labeled graph $G = (V, E)$ defined as follows:

(1) *Vertices*. There is a vertex labeled (T_k, R_j), $T_i \in \mathsf{TRS}_i$, $R_j \in RS_k$ only if the assignment of role R_j to task T_k does not violate workflow constraints. Thus, a vertex labeled (T_k, R_j) states that role R_j can be assigned to task T_k during workflow execution.

(2) *Edges*. There exists an arc connecting (T_i, R_j) to (T_h, R_k) only if the assignment of role R_j to task T_i and the assignment of role R_k to task T_h within the same workflow execution do not violate the constraints.

Algorithm 5.3 *The Role Planner*

INPUT: 1) The pruned workflow $Prun\_W = [\mathtt{TRS}'_1, \mathtt{TRS}'_2, \ldots, \mathtt{TRS}'_n]$, where $\mathtt{TRS}'_i = (T_i, (Upd\_RS_i, \succ_i), act_i)$
2) The pruned Constraint-Base $Prun\_\mathrm{CB}(W)$

OUTPUT: 1) FALSE if no role assignment satisfying the constraints can be generated; $RAG(W)$, otherwise.

1. Let n be the number of tasks in $Prun\_W$
2. *role_assignment*(1,[,])[a]
3. **If** $RAG(W)$ is empty: return FALSE

Procedure *role_assignment*(*current_task*, *ass_hyp*)

$j := current\_task$
Repeat
Let $Cand\_R := \{R_s \mid R_s \in Upd\_RS_j \wedge R_s$ is unmarked $\wedge\ \nexists R_k \in Upd\_RS_j$ such that $R_s \succ R_k\}$
If $Cand\_R \neq \emptyset$:
1. Let R_l be an element in $Cand\_R$, Mark R_l, $ass\_hyp[j] := R_l$
2. **If** $j < n$: *role_assignment*$(j+1, ass\_hyp)$
3. **If** $j = n$:
   a. **For** $i := 1$ to n:
      For $k := 1$ to act_i:
      Add $\mathtt{execute}_r(ass\_hyp[i], \mathtt{T}_i, k) \leftarrow$, $\mathtt{success}(\mathtt{T}_i, k) \leftarrow$ to $Prun\_\mathrm{CB}(W)$
   endfor
   b. **If** $Prun\_\mathrm{CB}(W)$ is consistent:
      $correct := true$, $i := 1$
      While $i \leq n$ and *correct*:
      Compute $Obliged\_Roles(T_i)$ and $Denied\_Roles(T_i)$
      If $Obliged\_Roles(T_i) \neq \emptyset$ and $ass\_hyp[i] \notin Obliged\_Roles(T_i)$:
      $correct := false$
      If $ass\_hyp[i] \in Denied\_Roles(T_i)$: $correct := false$
      $i := i + 1$
      endwhile
      If *correct*:
      For $i := 1$ to n: $v_i := (T_i, ass\_hyp[i])$
      Insert the path $p = \{v_1, \ldots, v_n\}$ into $RAG(W)$
      endif
   endif
4. **For** $m := 1$ to act_j:
   Remove $\mathtt{execute}_r(ass\_hyp[j], \mathtt{T}_j, m) \leftarrow$, $\mathtt{success}(\mathtt{T}_j, m) \leftarrow$ from $Prun\_\mathrm{CB}(W)$
endif
Until $\nexists$ an unmarked role in Upd_RS_j
Unmark the roles in Upd_RS_j

[a] '[,]' denotes the empty vector.

Fig. 8. Role planner.

In the following, given a label v, we use the notation $v.T$ to indicate the task appearing in v. Similarly, $v.R$ denotes the role appearing in v. Figure 8 reports the role planning algorithm.

The main step in Algorithm 5.3 is the recursive procedure *role_assignment* (step 2), which builds the role assignment graph of the input workflow. The candidate role assignments are stored into vector *ass_hyp*, which

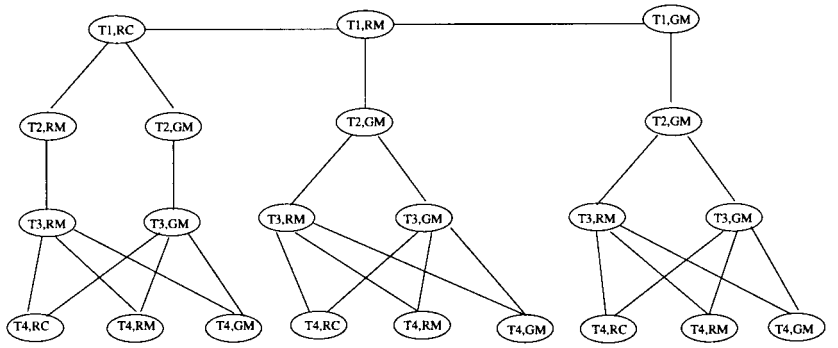

Fig. 9. The *RAG* generated by the role planner for the workflow in Example 2.1.

is incrementally built by recursively calling procedure *role_assignment*. Note that the first step of the **repeat-until** cycle of procedure *role_assignment* ensures that the various assignments are ordered in $RAG(W)$ according to the global role order. This ensures that at runtime a depth-first leftmost traversal of the tree always selects the assignment with the higher priority.

Example 5.3 Figure 9 shows the *RAG* generated by the role planner for the workflow in Example 2.2. Here and in the following example, we use abbreviations for role names. `RC` stands for `Refund Clerk`, `RM` for `Refund Manager`, whereas `GM` stands for `General Manager`. Note that the role planner considers all possible role assignments, including the case where no users belonging to the roles associated with the tasks are available. For instance, according to the *RAG* in Figure 9, if no user belonging to the role `Refund Clerk` is available, then task T_1 can be executed by role `Refund Manager` or `General Manager`. If T_1 is executed by `Refund Manager`, then T_2 must be executed by `General Manager` because of constraint C_2 in Example 3.1.

The termination of role planner is stated by the following theorem.

THEOREM 5.1 *Algorithm 5.3 terminates for each input CB and workflow role specification.*

We refer the reader to Appendix B for the formal proof.

Role plans are generated starting from the *RAG* returned by the role planner by selecting from the *RAG* each path p that contains one and only one node for each task in the workflow. Formally, given a workflow role specification $W = [\mathsf{TRS}_1, \ldots, \mathsf{TRS}_n]$, with $R\_Plans(RAG(W))$ we denote the set of path p in $RAG(W)$ such that (i) p has length n; and (ii) $\forall v_i, v_j \in p, i \neq j, v_i.T \neq v_j.T$.

Example 5.4 Considering the *RAG* in Figure 9, the following are paths in $R\_Plans(RAG(W))$: $\{(\mathsf{T}_1, \mathsf{RC}), (\mathsf{T}_2, \mathsf{RM}), (\mathsf{T}_3, \mathsf{RM}), (\mathsf{T}_4, \mathsf{RC})\}$, $\{(\mathsf{T}_1, \mathsf{RC}),$

(T_2, RM), (T_3, RM), $(T_4, RM)\}$, $\{(T_1, RC)$, (T_2, RM), (T_3, RM), $(T_4, GM)\}$, $\{(T_1, RC)$, (T_2, GM), (T_3, GM), $(T_4, RC)\}$, $\{(T_1, RC)$, (T_2, GM), (T_3, GM), $(T_4, RM)\}$, $\{(T_1, RC)$, (T_2, GM), (T_3, GM), $(T_4, GM)\}$.

The following theorem ensures that $R\_Plans(RAG(W))$ contains all, and only, the correct role assignments.

THEOREM 5.2 *Let* $W = [TRS_1, \ldots, TRS_n]$ *be a workflow role specification and let* $RAG(W)$ *be the corresponding RAG generated by the role planner. A path* $p = \{v_1, \ldots, v_n\}$ *belongs to* $R\_Plans(RAG(W))$ *iff the assignment of role* $v_i.R$ *to task* $v_i.T$, $\forall i = 1, \ldots, n$, *within the same workflow execution, does not invalidate workflow constraints.*

We refer the reader to Appendix B for the formal proof.

Note that the complexity of the role planner is exponential in the number of tasks in the workflow. More precisely, the overall worst-case complexity is $O(N_R^n)$, where N_R is the maximum number of roles associated with any task in the workflow and n is the number of tasks in the workflow. However, despite its worst-case behavior, we believe that the role planner is highly feasible in practice. The reason is twofold. First, both the number of tasks associated with a workflow and the number of roles associated with a task are usually not high. So the role planner's running time is acceptable in practice. Second, since the number of times a workflow is executed is typically very high (often in the range of tens of thousands), there is a considerable gain in performing role planning statically, before workflow starts, and then using it for all the instances of the workflow, rather than assigning roles at runtime without any planning.

5.3.2 *User Planning Phase.* User planning is performed by using the same strategies as in role planning.

It is quite common however for the number of users to be much larger than the number of roles. Typically, the number of users associated with a role increases as one goes down the role order hierarchy. So user planning is not always efficient, since an exhaustive check of all possible users who are allowed to execute tasks in the workflow is quite expensive because some roles may have a large number of authorized users. More precisely, the overall worst-case complexity in such case is $O((N_R \cdot N_U \cdot N_{act})^n)$, where N_R is the maximum number of roles associated with any task in the workflow; N_U is the maximum number of users associated with any role in the workflow; N_{act} is the maximum number of activations associated with any task in the workflow; and n is the number of tasks in the workflow. Thus, instead of using an exhaustive approach (as in the role planner), we use the following heuristics to do user planning. Consider the role assignment graph, each path $v_1, \ldots, v_n$ in $R\_Plans(RAG(W))$ represents a role plan. For each such plan we compute the maximum number of users required for each role R_i to execute the workflow tasks according to the plan. This set is denoted $U_{worstcase}(R_i)$. The worst-case scenario arises when

each task has to be executed by a different user and, moreover, each activation of a task must be executed by a different user. Suppose that each role R_i in a role plan executes a set of tasks $TASK_{R_i}$. Thus, $U_{worstcase}(R_i) = \Sigma_{T_j \in TASK_{R_i}} act_j$.

We perform user planning for only those roles R_i such that $U_{worstcase}(R_i) + const_i \leq U_i$ is not satisfied where $const_i$ is a safety factor. (This factor can be decided by the workflow designer based on several parameters such as the allowed number of simultaneous workflow executions, percentage of available users among the total users at any given time, etc.). Thus, user planning may not be executed for some roles in the role plans of $RAG(W)$. For such roles, users are assigned at runtime without any planning. Using this strategy, the overall worst-case complexity of the user planner is $O((k \cdot N_U(k) \cdot N_{act}(k))^n)$, where k denotes the number of roles for which we do user planning; $N_U(k)$ denotes the maximum number of users associated with any role for which we perform user planning; $N_{act}(k)$ is the maximum number of activations associated with any task for which we do user planning (that is, those tasks that have an associated role for which user planning is done); and n is the number of tasks in the workflow for which we do user planning. Note, however, that according to the definition of our heuristic, k will actually be small compared to the number of roles associated with the workflow. Moreover a role R_i for which user planning is executed is usually a role with a small number of associated users; otherwise the inequality $U_{worstcase}(R_i) + const_i \leq U_i$ would be satisfied and no user planning would have been performed for role R_i. We believe that user planning will be effective in practice.

A role plan in $RAG(W)$ may result in several user plans. From $RAG(W)$, we construct the *user-role assignment graph* ($URAG(W)$), which complements $RAG(W)$ with information about user plans. $URAG(W)$ is defined as follows:

Definition 5.2 (User-role assignment graph). Let $RAG(W)$ be a role assignment graph of a workflow role specification $W = [\mathsf{TRS}_1, \mathsf{TRS}_2, \ldots, \mathsf{TRS}_n]$, where each $\mathsf{TRS}_i = (T_i, (RS_i, >_i), act_i)$, $i = 1, \ldots, n$. The user-role assignment graph of W ($URAG(W)$) is a labeled graph $G = (V, E)$ defined as follows:

(1) *Vertices*. Vertices in $URAG(W)$ are of two different types:
   —There is a vertex labeled (T_i, R_k, UA), $UA = \{(u_1, 1), \ldots, (u_{act_i}, act_i)\}$, $u_j \in U_k$, $j = 1, \ldots, act_i$ only if (T_i, R_k) is a vertex of $RAG(W)$ and the assignment of user u_j to the j-th activation of task T_i does not violate the workflow constraints $j = 1, \ldots, act_i$.
   —There is a vertex labeled $(T_i, R_k, *)$ only if (T_i, R_k) is a vertex of $RAG(W)$ and user planning has not been executed for role R_k.

(2) *Edges*. Edges are of three different types, depending on the labels of the vertices they connect.

—There exists an arc connecting (T_i, R_k, UA) to (T_j, R_h, UA'), $UA = \{(u_1, 1), \ldots, (u_{act_i}, act_i)\}$, $UA' = \{(u'_1, 1), \ldots, (u'_{act}, act_j)\}$, $u_m \in U_k$, $m = 1, \ldots, act_i$, $u'_n \in U_h$, $n = 1, \ldots, act_j$ only if the assignment of user u_m to the m-th activation of task T_i, $m = 1, \ldots, act_i$ and the assignment of user u'_n to the n-th activation of task T_j, $n = 1, \ldots, act_j$ within the same workflow execution do not violate the constraints.

—There exists an arc connecting (T_i, R_k, UA) to $(T_j, R_h, *)$, $UA = \{(u_1, 1), \ldots, (u_{act_i}, act_i)\}$, $u_m \in U_k$, $m = 1, \ldots, act_i$, only if the assignment of user u_m to the m-th activation of task T_i, $m = 1, \ldots, act_i$, and the assignment of role R_h to each activation of task T_j within the same workflow execution do not violate the constraints.

—There exists an arc connecting $(T_i, R_k, *)$ to $(T_j, R_h, *)$ only if the assignment of role R_k to each activation of task T_i and the assignment of role R_h to each activation of task T_j within the same workflow execution do not violate the constraints.

Since user planning is performed using the same strategies as role planning, we do not report the details of the user planner algorithm here.

Example 5.5 Consider Example 2.2 again. Figure 10 represents a part of $URAG(W)$ corresponding to the RAG in Figure 9. $TASK_{\mathtt{GM}} = TASK_{\mathtt{RM}} = \{\mathtt{T}_2, \mathtt{T}_3\}$, $TASK_{\mathtt{RC}} = \{\mathtt{T}_1, \mathtt{T}_4\}$. Thus, $U_{worstcase}(\mathtt{GM}) = U_{worstcase}(\mathtt{RM}) = 3$, $U_{worstcase}(\mathtt{RC}) = 2$. $U_{\mathtt{GM}} = \{\mathtt{Ken}, \mathtt{Meg}\}$, $U_{\mathtt{RM}} = \{\mathtt{John}, \mathtt{Mary}, \mathtt{Tom}\}$, and $U_{\mathtt{RC}} = \{\mathtt{Bob}, \mathtt{Sam}, \mathtt{Matt}, \mathtt{Alice}\}$. Suppose that the safety factor is set equal to one for all the roles, so user planning is done for roles GM and RM only. Due to constraint $\mathtt{C}_5$, both user plans assign two different users to the two activations of task $\mathtt{T}_2$.

Similarly to RAGs, all the plans derivable from $URAG(W)$ are obtained by selecting the paths of length equal to the number of tasks in the workflow and containing one and only one node for each workflow task. We denote by $U\_Plans(URAG(W))$ the set of these paths.

5.4 Runtime Phase

The runtime phase is executed upon each task activation and termination. It consists of two subphases: the *task activation phase*, which is executed upon invocation of a task in the workflow, and the *task termination phase*, which is executed upon the termination of a task in the workflow.

Which user actually executes a given task is based not only on authorization constraints, but also on other factors, such as load balancing. Thus, when the k-th activation of a task T_j in a workflow W is invoked, the task activation phase is done to determine whether user u, executing T_j under role R_i, is authorized to do so, according to the the planning phase. As such, the authorization system presented in this paper acts as a server providing

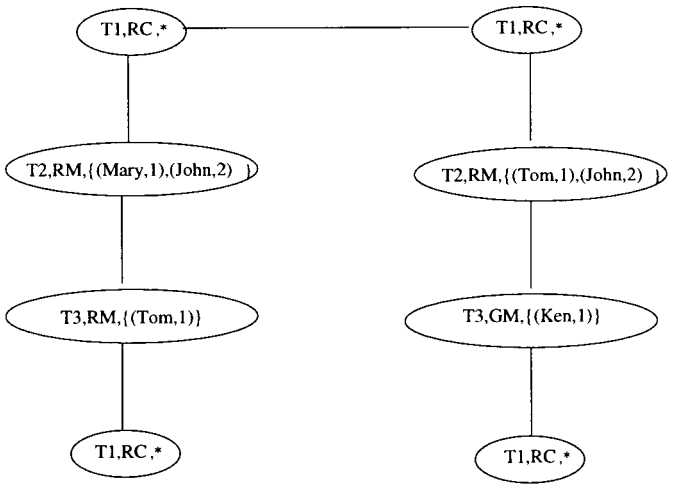

Fig. 10. The user-role assignment graph for the $RAG(W)$ in Figure 9.

information about authorization restrictions to the workflow management system. The authorization check entails verifying several conditions. First, user u must have the authorization to play role R_i, otherwise his/her request cannot be authorized. Then, if the number of activations of task T_j during workflow execution exceeds the number of activations specified in workflow role specification, the planner must be invoked again to verify that the execution of the current activation of task T_j by user u does not invalidate the current plan. Note that it is not necessary to re-execute the planner from the beginning; it can be invoked only for those task activations that have not yet been executed upon the k-th activation of task T_j. (For simplicity, we do not report this optimization in the algorithm.) If the user planning phase has been performed, user u must be among the users assigned to T_j, according to the user plan, and must not be *disabled* for the execution of the k-th activation of T_j by preceding task activations. The execution of the l-th activation of task T_m by a user u_n under role R_p disables u for the execution of the k- th activation of T_j under role R_i if the assignment of user u_n to the l-th activation of task T_m and the assignment of user u_i to the k-th activation of task T_j, within the same workflow execution, make the CB inconsistent. Checking these conditions ensure that tasks are executed by users according to the plans generated by the planning phase. To easily verify the above condition, $URAG(W)$ is pruned upon each task activation by removing the vertices corresponding to assignments disabled by the user/role that executed the task activation. Updating $URAG(W)$ is done by the task termination phase, described next. With such pruning, checking the above conditions is equivalent to verifying whether there exists in $URAG(W)$ a vertex labeled (T_j, R_i, UA) such that $\{(u_1, 1), \ldots, (u_{k-1}, k - 1), (u, k)\} \subseteq UA$, where u_i is the user who exe-

Algorithm 5.4 *The task activation algorithm*

INPUT: 1) A request $\langle u, R_i, T_j, k\rangle$ by user u to execute the k-*th* activation of task T_j under role R_i
2) The pruned workflow $Prun\_W = [\mathtt{TRS}'_1, \mathtt{TRS}'_2, \ldots, \mathtt{TRS}'_n]$, where $\mathtt{TRS}'_i = (T_i, (Upd\_RS_i, \succ_i), act_i)$
3) $Prun\_\mathrm{CB}(W)$
4) $URAG(W)$

OUTPUT: SUCCESS if the request can be authorized, FAILURE otherwise

1. **If** u is not authorized to play role R_i: return FAILURE
2. **If** $act_j < k$:
   Substitute act_j with k in $\mathtt{TRS}'_j$
   Execute the planning phase with $Prun\_W$ and $Prun\_\mathrm{CB}(W)$ as input
   If the planner returns FAILURE: return FAILURE
   endif
3. **If** the user planner has been activated for role R_i:
   Let u_i be the user that executed the i-*th* activation of task T_j, $i = 1, \ldots, k$-1
   If $\nexists$ in $URAG(W)$ a vertex labeled (T_j, R_i, UA) such that $\{(u_1, 1), \ldots, (u_{k-1}, k\text{-}1), (u, k)\} \subseteq UA$:
   return FAILURE
   endif
   else
   If $\nexists$ a vertex labeled $(T_j, R_i, *)$ in $URAG(W)$: return FAILURE
   Compute $Obliged\_Users(T_j)$ and $Denied\_Users(T_j)$
   If $Obliged\_Users(T_j) \neq \emptyset$ and $u \notin Obliged\_Users(T_j)$: return FAILURE
   If $Denied\_Users(T_j) \neq \emptyset$ and $u \in Denied\_Users(T_j)$: return FAILURE
   If $Prun\_\mathrm{CB}(W) \cup \{\mathtt{execute}_u(u, T_j, k) \leftarrow\}$ is inconsistent: return FAILURE
   endif
4. return SUCCESS

Fig. 11. The task activation algorithm.

cuted the i-th activation of T_j, $i = 1, \ldots, k - 1$. By contrast, if the user planning phase was not done for role R_i, it is necessary to verify that (i) role R_i was not disabled for the execution of the k-th activation of task T_j by the execution of tasks preceding T_j in the workflow; (ii) u belongs to the set of users, if any, for which it is mandatory to execute task T_j (that is, the set $Obliged\_Users(T_j)$); (iii) u does not belong to the set of users that when assigned to task T_j violate the workflow constraints (that is, the set $Denied\_Users(T_j)$); (iv) the execution of the k-th activation of T_j by u under role R_i does not make the CB inconsistent. This condition ensures that dynamic separation of duties is enforced.

If all the above checks succeed, user u is authorized to execute the k-th activation of task T_j, otherwise another user must be selected to execute task T_j, and all the steps described above are executed again.

An algorithm implementing the task activation phase is illustrated in Figure 11. It receives as input the request by user u to execute the k-th activation of task T_j under role R_i, the pruned CB, the pruned workflow, and the corresponding $URAG$. It returns SUCCESS if the request can be authorized, otherwise it returns FAILURE.

The task termination phase is performed upon task execution. During the task termination phase, information about task execution is inserted into $Prun\_\mathrm{CB}(W)$. Rules $\mathtt{execute}_r(R_i, T_j, k) \leftarrow$ and $\mathtt{execute}_u(u, T_j, k) \leftarrow$ are inserted into $Prun\_\mathrm{CB}(W)$ to record the fact that the k-th activation of task T_j was executed by user u under role R_i. Moreover, either the rule $\mathtt{success}(T_j, k) \leftarrow$ or $\mathtt{abort}(T_j, k) \leftarrow$ is inserted into $Prun\_\mathrm{CB}(W)$, depending on whether the k-th activation of task T_j aborts or successfully executes. The planner assigns roles/users to tasks under the assumption that all the tasks in the workflow successfully execute. If task T_j aborts, constraints involving abort predicates may no longer be satisfiable with the current plan. For instance, consider constraint $\mathtt{C}_6$ of Example 3.1 and suppose that task $\mathtt{T}_1$ is executed four times by user u and that all these executions abort. This implies that user u must be removed from the set of users that can execute task $\mathtt{T}_1$. Thus, if the abort of task $\mathtt{T}_1$ could cause the invalidation of the current plan, the planner is invoked again to determine whether the current plan needs to be modified. Note that it is not necessary to invoke the planner for all the task activations in the workflow, but only for those task activations that have not yet been executed upon the k-th activation of task T_j. Finally, the task termination phase updates $URAG(W)$ by removing the vertices corresponding to assignments that are disabled by the execution of the k-th activation of T_j by user u, that is, it removes those assignments that do not satisfy the workflow constraints, provided that u executes the k-th activation of task T_j. $URAG(W)$ is updated by means of a projection operator, formally defined as follows.

Definition 5.3 (URAG projection). Let W be a workflow role specification and let $URAG(W)$ be the corresponding user-role assignment graph generated by the user planner. Let v be a vertex in $URAG(W)$. The projection of $URAG(W)$ with respect to v, denoted $\Pi_v(URAG(W))$, is the graph obtained by removing from $URAG(W)$ each vertex v' such that $\nexists p \in U\_Plans(URAG(W))$ (cfr., Section 5.3.2) that contains both v and v'.

Intuitively, the projection operator is used to remove, from $URAG$, all the paths corresponding to assignments that are incompatible with the assignment of user u to the k-th activation of task T_j.

The algorithm that performs the task termination phase is shown in Figure 12.

Note that in the worst case, both the task termination and the task activation algorithm compute the model of the pruned CB and re-execute the planning phase. Thus, in the worst case, the complexity of these algorithms is proportional to the cost of performing the above operations. The complexity of the planning phase was discussed in Sections 5.3.1 and 5.3.2. The cost of computing the model is polynomial, since CB is a stratified logic program [Cadoli and Schaerf 1993]. Note, moreover, that the planning phase is re-executed using an incremental strategy, in that we

Algorithm 5.5 *The task termination algorithm*

INPUT: 1) The pruned workflow $Prun\_W = [\mathtt{TRS}'_1, \mathtt{TRS}'_2, \ldots, \mathtt{TRS}'_n]$, where $\mathtt{TRS}'_i = (T_i, (Upd\_RS_i, \succ_i), act_i)$
2) $Prun\_CB(W)$
3) A task T_j in $Prun\_W$
4) $URAG(W)$

OUTPUT: FAILURE if, upon the execution of task T_j, no role/user assignment satisfying the constraints can be generated, SUCCESS, otherwise

1. Let u be the user executing task T_j
2. Let R_i be the role under which u executed task T_j
3. Let k be the number of activations of task T_j during the workflow execution
4. Add $\mathtt{execute}_r(R_i, T_j, k) \leftarrow$ and $\mathtt{execute}_u(u, T_j, k) \leftarrow$ to $Prun\_CB(W)$
5. **If** task T_j successfully executes: Add $\mathtt{success}(T_j, k) \leftarrow$ to $Prun\_CB(W)$
   else
   Add $\mathtt{abort}(T_j, k) \leftarrow$ to $Prun\_CB(W)$
   Let M be the model of $Prun\_CB(W)$
   If $M(Prun\_CB(W)) \neq M$:
   Execute the planning phase with $Prun\_W$ and $Prun\_CB(W)$ as input
   If the planner returns FAILURE: return FAILURE
   endif
   endif
6. **If** the user planner has been activated for role R_i:
   Let u_i be the user that executed the *i-th* activation of task T_j, $i = 1, \ldots k$-1
   Let $v_1, \ldots, v_k$ be the vertex in $URAG(W)$ such that $v_i.T = T_j$, $v_i.R = R_i$, $\{(u_1, 1), \ldots, (u_{k-1}, k\text{-}1), (u, k)\} \subseteq v_i.UA^a$
   For $i = 1, \ldots, k$: $URAG(W) := \Pi_{v_i}(URAG(W))$
   else:
   $URAG(W) := \Pi_{(T_j, R_i, *)}(URAG(W))$
   Remove from $URAG(W)$ all the nodes v_i, $v_i.T = T_i$, $v_i.R = \{(u_1, 1), \ldots, (u_n, n)\}$ such that the assignment of u_i to the *i-th* activation of T_i and the assignment of u to the *k-th* activation of T_j within the same workflow execution make $Prun\_CB(W)$ inconsistent
   endif
7. return SUCCESS

[a] $v_i.UA$ denotes the set of pairs (u_k, k) in v_i.

Fig. 12. The task termination algorithm.

do not execute the planning phase for each task activation in the workflow but only for those activations not yet executed.

6. SYSTEM ARCHITECTURE

In this section, we discuss how our system, which we call *constraint analysis and enforcement module*, interacts with the WFMS in determining the role/user assignment for each task. Our system architecture is shown in Figure 13. Typically, a workflow is specified as a set of tasks and a set of dependencies among the tasks. Task dependencies can be specified on the basis of task primitives, such as begin, abort, and commit of a task, based on the outcome/result of a task, or on external parameters such as time

[Adam et al. 1998]. The WFMS is responsible for scheduling and synchronizing the various tasks within the workflow, in accordance with specified task dependencies, and for sending each task to the respective processing entity.

Although each processing entity may have its own role order and user-role assignment, these are all integrated and maintained at the central WFMS location. All the information concerning the role order and the user/role assignments are maintained by the WFMS.

Our constraint analysis and enforcement module determines the role assignments for each task in advance, that is, before the execution of the workflow. By contrast, user assignments are usually only partially computed before workflow execution, in that for some roles we determine the user assignments during workflow execution (cfr., Section 5.3.2). When a user submits a request to execute a task to the WFMS, the WFMS verifies whether the user can be authorized to execute the task according to the role/user assignments computed by the constraint analysis and enforcement module.

We have developed a prototype for our constraint analysis and enforcement module, which provides a graphical interface by which the user enters both workflow specification and workflow constraints and analyzes the results of each phase of our methodology. The user enters the constraints by means of an interactive interface, the constraints are then translated by the system into rules in our constraint specification language. The constraint specification language was implemented using the CORAL system [Ramakrishnan et al. 1994]. CORAL is a deductive system that supports a rich declarative language and an interface to C++. Coral rules are syntactically similar to Prolog rules, and provide support for both stratified negation and aggregate predicates. CORAL uses bottom-up evaluation, with a wide variety of optimization strategies specified by the programmer.

7. CONCLUSIONS

An organization's security policies, such as separation of duties, are often expressed as constraints on users and roles. Current authorization models, however, do not support the specification and evaluation of such constraints. In this paper, we proposed a language for expressing such constraints and classified the various types of constraints—static, dynamic, and hybrid—based on the time at which they can be evaluated. While static constraints can be evaluated before execution of the workflow, dynamic constraints can only be evaluated during execution of the workflow. Hybrid constraints can be partially evaluated before execution. In this paper, we also devised algorithms to check the consistency of the constraints in order to consistently assign roles and users to tasks in the workflow. To minimize the complexity of consistency checking during execution of the workflow, we presented an approach to generate assignments of roles and users to tasks that satisfies authorization constraints. If tasks are executed at runtime, according to the plan, several dynamic constraints are automati-

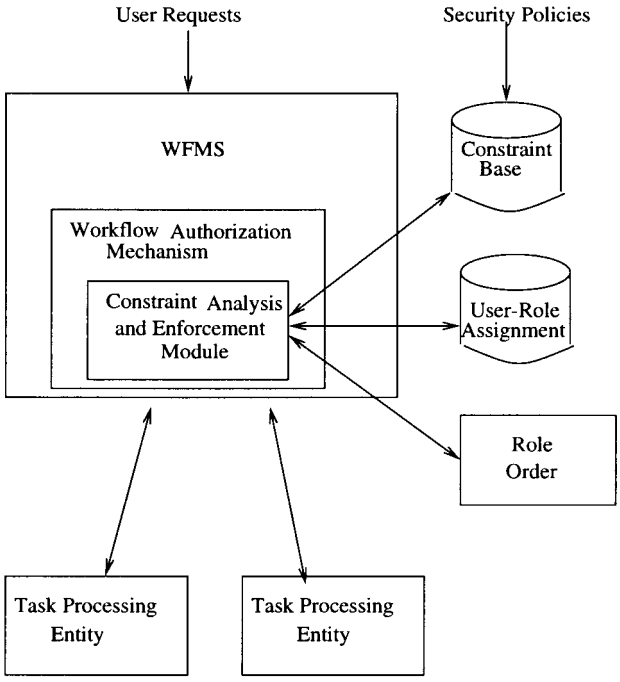

Fig. 13. The system architecture.

cally satisfied. Thus our approach requires checking only a very few constraints during execution of the workflow.

The work presented in this paper can be extended in several directions. In particular, future work includes modeling temporal and event-based constraints, support for more complex models of roles, along the lines discussed by Sandhu [1996], and the use of our system in heterogeneous [Bonatti et al. 1996] and distributed environments. An important question that also needs to be addressed is optimization of the role planner. In our approach, we performed an exhaustive search to determine all role assignments to tasks that do not violate constraints. Such a search is performed just once, before the workflow is executed. All executions of the same workflow reuse the same role assignment plan, and so the cost of planning is amortized over several executions. It would, however, be useful to devise heuristics to reduce search space. We plan to extensively investigate optimization techniques for role planning based on constraint analysis. For instance, workflow constraints can be analyzed before executing the role planner, to identify those tasks not involved in any constraint. Roles can be assigned to such tasks at runtime without any planning.

We also plan to perform an exhaustive performance evaluation of user planning and to investigate the use of various heuristics in selecting the set of roles for which a user planner should be activated.

Another important issue is the management of change in workflow specification. When a user and/or role is inserted/removed from the list of

users/roles assigned to a given task, planning must be modified accordingly. A possible solution is to re-execute the planning phase each time there is an update to the workflow specification. However, more efficient solutions can be devised that allow incremental maintenance of role and user planning by modifying only those parts of planning that are actually affected by the updates. We plan to investigate the use of such incremental strategies in the near future.

Finally, a major extension is the concurrent execution of tasks. In this paper we make the assumption that all the tasks in a workflow are executed sequentially, according to a specified total order. Relaxing such an assumption implies that only a partial execution order may be specified for tasks in a workflow. When task executions are partially ordered, some additional restrictions arise on the constraints one can specify. For instance, if two tasks can be executed concurrently, it is impossible to constraint the role/user assignment of a task to the abortion/success of the other task. We plan to devise methods to automatically determine such inconsistencies, based on the specified task execution order.

APPENDIX

A. ATOMS AND LITERALS

In the following, we list the atoms and literals that can be specified in our language.

Let $ut, ut' \in UT$, $rt, rt' \in RT$, $tt, tt' \in TT$, $ct \in CT$, and $nt, nt' \in \mathbb{N}T$, then we say,

specification atoms = {role(rt, tt), user(ut, tt), belong(ut, rt), glb(rt, tt), lub(rt, tt), $rt > rt'$};

execution atoms = {execute$_u$(ut, tt, nt), execute$_r$(rt, tt, nt), abort(tt, nt), success(tt, nt)};

planning atoms = {cannot_do$_u$(ut, tt), cannot_do$_r$(rt, tt), must_execute$_u$(ut, tt), panic, must_execute$_r$(rt, tt), statically_checked(ct)};

comparison atoms = {$ut = ut'$, $rt = rt'$, $tt = tt'$, $nt = nt'$, $nt < nt'$, $nt \leq nt'$, $nt > nt'$, $nt \geq nt'$}.

Similarly:

specification literals = {role(rt, tt), not role(rt, tt), belong(ut, rt), not belong(ut, rt), user(ut, tt), not user(ut, tt), glb(rt, tt), not glb(rt, tt), lub(rt, tt), not lub(rt, tt) $rt > rt'$, $rt \not> rt'$};

execution literals = {execute$_u$(ut, tt, nt), not execute$_u$(ut, tt, nt), execute$_r$(rt, tt, nt), not execute$_r$(rt, tt, nt),

abort(tt, nt), not abort(tt, nt), success (tt, nt), not success(tt, nt)};

planning literals = {cannot_do$_u$(ut, tt), not cannot_do$_u$(ut, tt), cannot_do$_r$(rt, tt), not cannot_do$_r$(rt, tt), must_execute$_u$(ut, tt), not must_execute$_u$(ut, tt), must_execute$_r$(rt, tt), not must_execute$_r$ (rt, tt), panic, not panic, statically_checked (ct), not statically_checked(ct)};

comparison literals = $\{ut = ut', rt = rt', tt = tt', nt = nt', ut \neq ut', rt \neq rt', tt \neq tt', nt \neq nt', nt < nt', nt \not< nt', nt \leq nt', nt \not\leq nt', nt > nt', nt \not> nt', nt \geq nt', nt \not\geq nt'\}$.

Moreover, let W be a conjunction of literals, x be a variable bound in W, and $nt \in NT$, then:

aggregate literals = {count(W, nt), not count(W, nt), avg(x, W, nt), not avg (x, W, nt), min(x, W, nt), not min(x, W, nt), max(x, W, nt), not max(x, W, nt), sum(x, W, nt), not sum(x, W, nt)}.

B. PROOFS

Proof of Proposition 4.1

A program P is stratified if its *extended dependency graph* does not contain any cycle involving an edge labeled "not" [Ullman 1989], where the extended dependency graph of a program P is a graph whose nodes are the predicates that appear in the heads of the rules of P. Given two nodes p_1 and p_2, there is a direct edge from p_1 to p_2 if and only if predicate p_2 occurs positively or negatively in the body of a rule whose head predicate is p_1. The edge (p_1, p_2) is marked with a "not" sign if and only if there exists at least one rule r with head predicate p_1 such that p_2 occurs negatively in the body of r. By Definition 4.7, the CB associated with a given workflow consists of a set of explicit assignment and integrity rules. Let us consider each of these rules. By Definition 4.1, explicit rules have an empty body and a specification or execution atom as head. Thus, they cannot form any cycle in the extended dependency graph of a CB. By Definitions 4.2 and 4.4, assignment and integrity rules have planning predicates as head and a conjunction of specification, execution, comparison literals, and aggregate atoms as body. Since the predicates that can appear in the head of an assignment or integrity rule are disjoint from the predicates that can appear in its body, they cannot form any cycle in the extended dependency graph. Hence, the extended dependency graph associated with a CB does not contain any cycle. Thus the CB is stratified.

Proof of Theorem 5.1

To prove the termination of the role planner, it is sufficient to prove the termination of procedure *role_assignment* (step 2). Consider procedure *role_assignment*. Suppose that it never terminates. Procedure *role_assignment* consists of a **repeat-until** loop that contains a recursive call to procedure *role_assignment* itself. *role_assignment* is called by the role planner in step 2 with 1 and [,] as parameters. Consider the execution of *role_assignment*(1,[,]). *role_assignment*(1,[,]) selects a role R_l from Upd_RS_1, marks R_l, and inserts it into *ass_hyp*[1]. Then *role_assignment*(2,*ass_hyp*) is invoked. This call selects a role from Upd_RS_2, marks it, inserts it into *ass_hyp*[2], and invokes *role_assignment*(3,*ass_hyp*). This process is iteratively executed till *role_assignment*(*n*,*ass_hyp*) is invoked, where n is the number of tasks in the workflow. At this point no more calls of procedure *role_assignment* are executed. Consider *role_assignment*(*n*,*ass_hyp*). It consists of an outer **repeat-until** loop which is iterated until no unmarked roles exist in Upd_RS_n. Consider the **repeat-until** loop. Termination of steps 1 and 2 is straightforward. Consider step 3. Step 3a terminates, since it consists of two **for** loops that operate on a finite number of elements (the number of tasks in a workflow and the number of activations of a given task, respectively). Consider step 3b. The **while** loop has a condition on variables i ($i \leq n$) and *correct* (*correct* = *true*). Variable i is set equal to 1 before entering the **while** loop and increased at each iteration. Then, after at most n iterations, the condition on i will evaluate *false* and the loop will terminate. The **for** loop terminates, since the number of tasks in a workflow is finite. Thus, since checking the consistency of a CB is a finite process, step 3b terminates. The **for** loop in step 3 terminates, since the number of tasks in a workflow is finite. Thus, step 3 terminates. Termination of step 4 is straightforward, since the number of activations of a given task in a workflow is finite. Now consider the outer **repeat-until** loop. At each iteration of the **repeat-until** loop, a role in Upd_RS_n is marked. Thus, since the number of roles associated with a task in a workflow is finite, *role assignment*(*n*,*ass_hyp*) halts, after a finite number of iterations, unmarking all the roles in Upd_RS_n. Thus, *role_assignment* ($n - 1$,*ass_hyp)* resumes its execution by selecting a role from Upd_RS_{n-1}, different from the one selected in the preceding iteration. The selected role is marked and inserted into *ass_hyp*[$n - 1$]. Then, *role_assignment*(*n*,*ass_hyp)* is invoked again. When *role_assignment*(*n*,*ass_hyp*) halts, another role is selected from Upd_RS_{n-1}, marked and assigned to *ass_hyp*[$n - 1$]. Then *role_assignment*(*n*,*ass_hyp*) is invoked again. This process is iteratively executed until no unmarked roles exist in Upd_RS_{n-1}. Since the number of roles associated with a task is finite, *role_assignment*($n - 1$,*ass_hyp*) halts after a finite number of iterations by unmarking all the roles in Upd_RS_{n-1}. Based on the same logic, we can prove the termination of all the recursive calls of procedure *role_assign-*

ment up to *role_assignment*(1,[,]), that is, the call directly invoked by the role planner. Thus, the algorithm terminates.

Proof of Theorem 5.2

We first prove the *if* part of the thesis. Let $p = \{v_1, \ldots, v_n\}$ be a path in $R\_Plans(RAG(W))$. We have to prove that the assignment of role $v_i.R$ to task $v_i.T$, within the same workflow execution, does not violate workflow constraints, $i = 1, \ldots, n$. We suppose that the thesis does not hold and derive a contradiction. Paths in $R\_Plans(RAG(W))$ are those inserted into $RAG(W)$ by step 3 of the algorithm. It is easy to see that proving the thesis by contradiction is equivalent to proving that there exists a path $p = \{v_1, \ldots, v_n\}$ added to $RAG(W)$ by step 3 of the role planner, and that either (1) $CB' = Prun\_CB(W) \cup \{\texttt{execute}_r(v_i.R, v_i.T, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\} \cup \{\texttt{success}(v_i.R, v_i.T, k) \leftarrow \mid i = 1, \ldots, n, \ k = 1, \ldots, act_i\}$ is inconsistent; or (2) there exists a label v_i in path p such that (i) $Obliged\_Roles(v_i.T) \neq \emptyset$ and $v_i.R \notin Obliged\_Roles(v_i.T)$ or (ii) $v_i.R \in (Upd\_RS_i \cap Denied\_Roles(T_i))$. Suppose (1) holds. The role planner incrementally builds the paths to be added to $RAG(W)$ by recursively calling procedure *role_assignment*. When a candidate role assignment (stored in *ass_hyp*) is constructed (that is, when the task currently examined by procedure *role_assignment* is equal to the last task of the workflow), procedure *role_assignment* adds (step 3a): $\{\texttt{execute}_r(v_i.R, v_i.T, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\} \cup \{\texttt{success}(v_i.T, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\}$ to $Prun\_W$. It then checks for the consistency of the updated $Prun\_\mathrm{CB}(W)$, and the path is added to $RAG(W)$ only if the check succeeds (step 3b). Thus, since the updated $Prun\_\mathrm{CB}(W)$ is equal to CB′, we have a contradiction.

Now suppose that (2) holds. First consider case (2i); that is, suppose there exists a label v_i in p such that $Obliged\_Roles(v_i.T) \neq \emptyset$ and $v_i.R \notin Obliged\_Roles(v_i.T)$. Procedure *role_assignment*, after checking the consistency of the updated $Prun\_\mathrm{CB}(W)$ (step 3b), computes the sets *Obliged_Roles* and *Denied_Roles* for each task in the workflow. For each task T_i such that $Obliged\_Roles(T_i) \neq \emptyset$, it checks whether the role assigned to T_i in the candidate path does not belong to $Obliged\_Roles(T_i)$. If the check succeeds, variable *correct* is set to *false*. Thus, the subsequent **if** statement is not executed and the path is not added to $RAG(W)$. Now suppose that case (2ii) holds, that is, there exists a label v_i in p such that $v_i.R \in Upd\_RS_i \cap Denied\_Roles(T_i)$. In step 3b, procedure *role_assignment* checks, for each task T_i, whether the role assigned to T_i in the candidate path belongs to $Denied\_Roles(T_i)$. Since the role assigned to T_i in the candidate path belongs to Upd_RS_i (by step 1), this is equivalent to checking that the role assigned to T_i in the candidate path belongs to $(Denied\_Roles(T_i) \cap Upd\_RS_i)$. If the check succeeds, variable *correct* is set to *false*. This implies that the subsequent **if** statement is not executed

and the candidate path is not added to $RAG(W)$. Thus, in both the above cases we have a contradiction.

Now consider the other part of the implication. As in the previous case, we suppose that the implication does not hold and derive a contradiction. We suppose that there exists a set of roles $S = \{R_1, \ldots, R_n\}$ such that the assignment of R_i to task T_i within the same workflow execution does not violate constraint $i = 1, \ldots, n$ and that path $p = \{v_1, \ldots, v_n\}$ such that $v_i.T = T_i, v_i.R = R_i, i = 1, \ldots, n$ does not belong to $R\_Plans(RAG(W))$. This is equivalent to supposing that there exists a set of roles $S = \{R_1, \ldots, R_n\}$ such that $R_i \in Upd\_RS_i$, $R_i \in Obliged\_Roles(T_i)$, if $Obliged\_Roles(T_i) \neq \emptyset$, $R_i \in Upd\_RS_i \backslash Denied\_Roles(T_i)$, otherwise $CB' = Prun\_CB(W) \cup \{\mathsf{execute}_r(R_i, T_i, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\} \cup \{\mathsf{success}(T_i, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\}$ is consistent; the path p $= \{v_1, \ldots, v_n\}$, such that $v_i.R = R_i$, $v_i.T = T_i$, $i = 1, \ldots, n$ is not added to $RAG(W)$ by the role planner. Consider the role planner. In step 2, the call *role_assignment*(1,[,]) is executed. Next, consider the execution of *role_assignment*(1,[,]). In step 1, a role is selected from $Cand\_R \subseteq Upd\_RS_1$ and stored in the first component of *ass_hyp* (that is, the vector storing the current candidate path). We can distinguish two cases on the basis of whether R_1 is selected or not. Consider the case in which R_1 is selected. Then, if the workflow contains more than one task, that is, if $j = 1 < n$, *role_assignment*(2,*ass_hyp*) is executed, where *ass_hyp* contains, in the first component, role R_1. Consider the execution of *role_assignment*(2,*ass_hyp*). In step 1, a role is selected from $Cand\_R \subseteq Upd\_RS_2$ and stored in the second component of *ass_hyp*. Again, we can distinguish two cases, depending on whether role R_2 is selected or not. As in the previous case, we first look at where role R_2 is selected. If the workflow contains more than two tasks, that is, if $j = 2 < n$, *role_assignment*(3,*ass_hyp*) is executed. This process is iteratively executed until the last task in the workflow is reached, that is, until *role_assignment*(*n*,*ass_hyp*) is invoked. From the way *ass_hyp* was built, it contains in the i-th position role R_i, $i = 1, \ldots, n - 1$. Consider the execution of *role_assignment*(*n*,*ass_hyp*). In step 1, a role is selected from $Cand\_R \subseteq Upd\_RS_n$ and stored in the n-th component of *ass_hyp*. As in the previous cases, we first examine where role R_n is selected first. Then, since $j = n$, procedure *role_assignment* adds $\{\mathsf{execute}_r(R_i, T_i, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\} \cup \{\mathsf{success}(T_i, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\}$ to $Prun\_W$ (step 3a). By hypothesis, the updated $Prun\_W$ is consistent, then the subsequent **while** loop is executed. It is easy to verify that at the end of the **while** loop, variable *correct* has value *true* only if for each R_i, $i = 1, \ldots, n$, in *ass_hyp*: (1) $R_i \in Obliged\_Roles(T_i)$, if $Obliged\_Roles(T_i) \neq \emptyset$; (2) $R_i \notin Upd\_RS_i \cap Denied\_Roles(T_i)$. The above conditions are true by hypothesis, thus the subsequent **if** statement is executed and the

path $p = \{v_1, \ldots, v_n\}$, $v_i.T = T_i$, $v_i.R = R_i$, $i = 1, \ldots, n$ is added to $RAG(W)$, and we have a contradiction.

Suppose now that role R_n is not selected during the first iteration of the **repeat-until** loop in *role_assignment*(n,*ass_hyp*). This means that a role $R'_n \in Cand\_R \subseteq Upd\_RS'_n$, $R_n \neq R'_n$ is inserted into $ass\_hyp[n]$. Role R'_n is marked, and $\{\mathsf{execute}_r(R_i, T_i, k) \leftarrow \mid i = 1, \ldots, n-1, k = 1, \ldots, act_i\} \cup \{\mathsf{success}(T_i, k) \leftarrow \mid i = 1, \ldots, n, k = 1, \ldots, act_i\} \cup \{\mathsf{execute}_r(R'_n, T_n, k) \leftarrow k = 1, \ldots, act_n\}$, are added to $Prun\_\mathrm{CB}(W)$. Then, in step 3, $\{\mathsf{execute}_r(R'_n, T_n, k) \leftarrow k = 1, \ldots, act_n, \mathsf{success}(T_n, k) \leftarrow, k = 1, \ldots, act_n\}$ are removed from $Prun\_\mathrm{CB}(W)$ and another role, among the unmarked roles in Upd_RS_n, is selected. This process is iterated until no unmarked roles exist in Upd_RS_n, then all the roles in Upd_RS_n are unmarked and *role_assignment*(n,*ass_hyp*) terminates. Thus, since by hypothesis, $R_n \in Upd\_RS_n$, after a finite number of iterations R_n will be selected and assigned to $ass\_hyp[n]$.

Now suppose that role R_{n-1} is not assigned to $ass\_hyp[n-1]$ during the first iteration of procedure *role_assignment*($n-1$,*ass_hyp*), where $ass\_hyp[i] = R_i$, $i = 1, \ldots, n-2$. Using the same reasoning we used for R_n, we can show that, after a finite number of iterations, role R_{n-1} is selected and assigned to $ass\_hyp[n-1]$. In a similar way, we can apply the same steps that we illustrated for R_n to all the roles R_i, $i = n-2, \ldots, 1$. Thus the claim holds.

REFERENCES

ADAM, N., ATLURI, V., AND HUANG, W. K. 1998. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst. 10*, 2, 131–158.

BONATTI, P., SAPINO, M., AND SUBRAHMANIAN, V. S. 1996. Merging heterogeneous security orderings. In *Proceedings of the Conference on Computer Security* (ESORICS 96, Rome, Italy), E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Eds. Springer-Verlag, New York, NY, 183–197.

CADOLI, M. AND SCHAERF, M. 1993. Complexity results for non-monotonic logics. *J. Logic Program. 17*.

CHANG, S., POLESE, G., THOMAS, R., AND DAS, S. 1997. A visual language for authorization modeling. In *Proceedings of the IEEE Symposium on Visual Languages* (VL97, Capri, Italy). IEEE Computer Society Press, Los Alamitos, CA.

CLARK, D. AND WILSON, D. 1987. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA). IEEE Computer Society Press, Los Alamitos, CA, 184–194.

DAS, S. 1992. *Deductive Databases and Logic Programming*. Addison-Wesley, Reading, MA.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming* (Cambridge, MA). MIT Press, Cambridge, MA, 1070–1080.

GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. 1995. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases 3*, 2 (Apr. 1995), 119–153.

JONSCHER, D., MOFFET, J., AND DITTRICH, K. 1994. Complex subjects or the striving for complexity is ruling our world. In *Database Security VII: Status and Prospects*. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 19–37.

LLOYD, J. W. 1984. *Foundations of Logic Programming*. Springer-Verlag, New York, NY.

LOTUS CORPORATION, 1996. *Lotus Notes Administrator's Reference Manual, Release 4*. Lotus Publ. Corp., Cambridge, MA.

MEDINA-MORA, R., WONG, H., AND FLORES, P. 1993. ActionWorkflow as the enterprise integration technology. *IEEE Data Eng. Tech. Bull. 16*, 2, 49–52.

NYANCHAMA, M. AND OSBORN, S. 1993. Role-based security, object oriented databases and separation of duty. *SIGMOD Rec. 22*, 4 (Dec. 1993), 45–51.

NYANCHAMA, M. AND OSBORN, S. 1996. Modeling mandatory access control in role-based security systems. In *Database Security IX: Status and Prospects*. Elsevier North-Holland, Inc., New York, NY, 129–144.

Proceedings of the 1st (1996) ACM Workshop on Role-Based Access Control. ACM Press, New York, NY.

RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1994. The coral deductive system. *VLDB J. 3*, 2, 161–210.

SANDHU, R. 1991. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects*. Elsevier North-Holland, Inc., New York, NY, 179–189.

SANDHU, R. 1996. Role hierarchies and constraints for lattice-based access controls. In *Proceedings of the Conference on Computer Security* (ESORICS 96, Rome, Italy), E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Eds. Springer-Verlag, New York, NY, 65–79.

SANDHU, R., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. 1996. Role-based access control models. *IEEE Comput. 29*, 2 (Feb.), 38–47.

THOMAS, R. AND SANDHU, R. 1997. Task-based authorization controls (TBAC): Models for active and enterprise-oriented authorization management. In *Proceedings of the 11th IFIP Working Conference on Database Security* (Lake Tahoe, CA). Chapman & Hall, Ltd., London, UK, 136–151.

ULLMAN, J. 1989. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Inc., New York, NY.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38*, 3 (July 1991), 619–649.

Received: September 1997; revised: May 1998; accepted: October 1998