Managing Workflow Authorization Constraints through Active Database Technology

Fabio Casati, Silvana Castano[1], MariaGrazia Fugini[2]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-156
November 29th, 2000*

E-mail: casati@hpl.hp.com, castano@dsi.unimi.it, fugini@elet.polimi.it

workflows, authorization constraints, active rules

The execution of workflow processes requires authorization models and tools for enforcing the assignment of tasks to (human or automated) agents according to the security policy of the organization. The paper presents an advanced role-based authorization model for workflow processes, extended with organizational levels and authorization constraints. Roles and organizational levels are organized into hierarchies, to facilitate the assignment of tasks to agents. In addition, constraints are introduced to specify instance-dependent, time-dependent, and history-dependent authorizations. Authorization constraints are specified in terms of active rules to be executed on top of the authorization base, where play and execute authorizations as well as role and level hierarchies are properly stored. Besides enforcing authorization constraints, active rules are used also for authorization management, to enforce authorization derivation along role/level hierarchies. The WfMS then determines authorized agents on the basis of the contents of the authorization base, suitably maintained by the active rules defined in the system. In order to better illustrate the model and concepts included in the paper and to demonstrate the feasibility of the approach, we also present the implementation of the proposed model within the WIDE workflow management system.

* Internal Accession Date Only　　　　Approved for External Publication
[1] Dipartimento di Scienze dell'Informazione – Universit`a di Milano, I-20135 Milano, Italy
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, I-20133 Milano, Italy
© Copyright Hewlett-Packard Company 2000

1 Introduction

Workflows processes are complex activities involving the coordinated execution of several tasks by different executing agents, in order to reach a common objective [18]. Workflow Management Systems (WfMSs) are software applications that support both the design of workflows and their execution. Workflow processes are inherently distributed, and are co-operatively executed by users and applications, possibly spanning beyond organizational boundaries. Consequently they are characterized by general security requirements of such kind of processes. Discretionary models, based on the notion of task-based authorization, have been studied [30] to provide a flexible and adaptable access control paradigm for distributed processes. In addition to these aspects, workflows present some peculiar security requirements that have to be considered. In particular, a security-relevant aspect of workflows is related to the assignment of tasks to (human or automated) agents in the system, which is automatically performed by the WfMS, according to properly defined models and rules [5]. A basic role-based authorization model [31] is commonly adopted in most commercial and research workflow systems, to comply with general requirements of organizational security policies. In fact, these policies often express task assignment in terms of roles (i.e., job functions or characteristics of the required application) rather than in terms of specific individuals, reducing the number of authorizations necessary in the system and simplifying their maintenance. However, a role-based model alone is not sufficient to meet all the requirements of security policies of the organization. In particular, such policies often demand capabilities of expressing and enforcing authorization constraints, such as the well known separation of duties [31].

Therefore, more advanced role-based models are necessary, together with supporting technology, in order to enable the definition of authorization constraints in the WfMS, and to be able to implement the many different security policies of an organization.

In this paper, we present an advanced authorization model for the assignment of tasks to roles, organizational levels, and agents. Roles and organizational levels are organized into hierarchies, to facilitate the assignment of tasks to agents. Authorizations for agents to play roles/levels and for roles/levels to execute tasks can be defined for all instance of a given workflow process,independent of time and history in the system. In addition, the model enables the definition of instance-dependent, time-dependent, and history-dependent authorizations in the form of constraints: authorizations can be modified depending on the state or history of a workflow instance, on time, or on the content of process-specific data. Authorization constraints are expressed as Event-Condition-Action (ECA) rules, where the event part denotes when an authorization may need to be modified, the condition part verifies that the occurred event actually requires modifications of authorizations, and determines

the involved agents, roles, tasks, and processes, while the action part enforces authorizations and prohibitions. Indeed, ECA rules are a very flexible mean for defining authorization constraints, since they enable the definition of when and how an authorization should be modified.

Besides being a natural mean for defining authorization constraints, active rules and active database technology can be exploited for the implementation of the model. In our approach, authorization constraints are defined and implemented by means of ECA rules, executed on top of a suitable authorization base, where play and execute authorizations as well as role and level hierarchies are stored. Active database technology supports the definition and execution of ECA rules which are sensitive to several different event types (such as data or temporal events, as well as notifications from external applications), query the database in the condition part and, depending on the result of the query, modify the database state or activates external applications. Active rules introduce a significant increase of the expressive power of database languages, resulting in the enhancement of processing capabilities within the database servers [13, 36]. They are supported by most relational database systems, including DB2, Illustra, Informix, Ingres, Oracle, RDB, Sybase and others. Active rules in these systems support basic functionality and are inspired by the SQL3 standard [16]. Since the majority of WfMS are developed on top of a commercial DBMS with active capabilities, ECA rule-based authorization constraints can be implemented within most existing WfMSs.

Besides authorization constraint enforcement, active rules can be exploited for managing authorization inheritance along the role and level hierarchies of the model: as an authorization for a role or level is granted (revoked), suitable active rules ensure that the same authorization is granted (revoked) to their ancestors. In this paper, we propose the use of active rules both for the definition of the operational semantics of authorization inheritance and as a suitable implementation mechanism.

Hence, active rules suitably maintain the authorization base: at run-time, when a task is scheduled for execution, the WfMS accesses the authorization base and determines which are the roles, organizational levels, and eventually the agents authorized to execute the newly activated task, and assign it to those agents. In order to better illustrate the model and concepts included in the paper and to demonstrate the feasibility of the approach, we will also present the implementation of the proposed model within the WIDE WfMS [20].

The paper is organized as follows. Section 2 introduces basic workflow concepts and presents a reference example, used throughout the paper. Section 3 presents the advanced role-based authorization model, based on authorization constraints. Section 4 shows how ECA rules can be exploited to define and enforce authorization constraints and to manage

derivation of authorization; Section 5 presents the implementation of the models and concepts introduced in the paper on top of the WIDE WfMS; Section 6 discusses the related work, and finally Section 7 outlines concluding remarks and future research directions.

2 Basic workflow concepts

This section introduces basic workflow concepts, according to the model and terminology defined by the Workflow Management Coalition [24, 34, 33].

A *workflow process definition* (or simply *workflow schema*) is the formal representation of a business process. A workflow schema is composed of subprocesses and of elementary activities (tasks) that collectively achieve the business goal. Activities are organized into a directed graph (the *flow structure*), that defines the order of execution among the activities in the process. Arcs in the graph may be labeled with transition predicates defined over process data, meaning that as an activity is completed, tasks connected to outgoing arcs are executed only if the corresponding transition predicate evaluates to true. A *process instance* or *case* is an enactment of a workflow schema. A schema may be instantiated several times, and several instances may be concurrently running. WfMSs support case execution by scheduling tasks (as defined by the flow structure) and by assigning them for execution to human or automated agents.

A sample process, related to the evaluation of a medical insurance request, is depicted in Figure 1. An instance (or *case*) of this workflow is started as a new medical insurance request is received. First, data are collected and inserted in electronic format (`Data collection` task). Then, the `Evaluation` task checks whether a medical advice is needed, and in this case the request is routed to an insurance doctor, who will write a medical report on the applicant's physical conditions (`Medical Examination` task). Otherwise, the request is directly forwarded to the evaluation manager for the final decision on the customer's request (`Decision` task). If the request is accepted (`status="accepted"`), a dossier is prepared for the new customer (`Customer dossier preparation` task), otherwise the rejection is notified to the applicant (`Notification of rejection` task). Finally, the insurance documents are issued to the interested offices and filed (`Issuing` and `Filing` tasks).

A process may create and access several types of data. The WfMC identifies three types of workflow data:

- *workflow relevant data* includes typed data created and used by a process instance; these data can be made available to subprocesses and activities, and can be accessed by the WfMS in order to evaluate transition predicates.

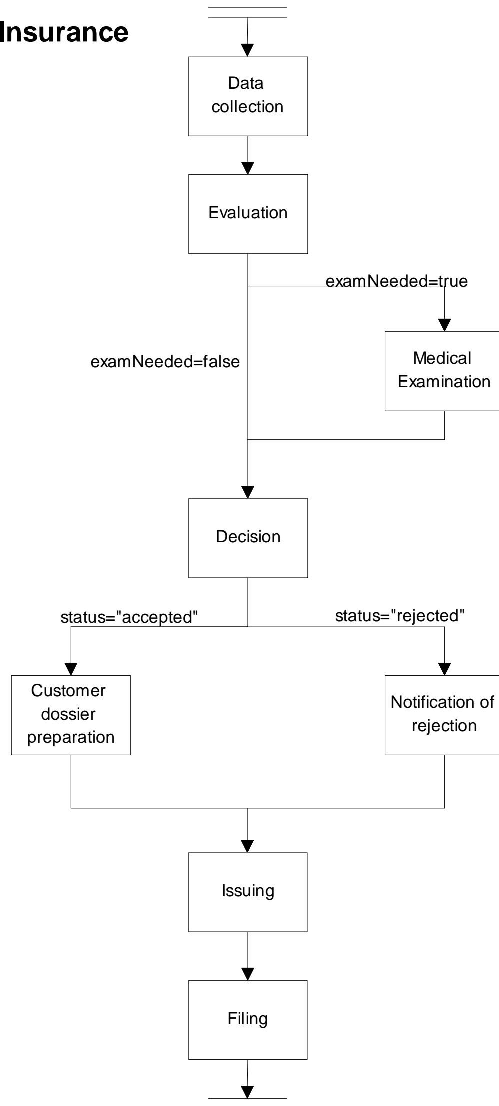

Figure 1: The `Medical Insurance` process

- *Application data* are application specific, must be processed with external tools (such as Microsoft Word©) and cannot be accessed by the WfMS, although the system may control and restrict accesses to them.
- *System and environmental data* are maintained by the local system environment or by the WfMS itself, and can be used to evaluate transition predicates.

Activities are typically executed atomically with respect to workflow relevant data, and data modifications are made visible as the task is completed.

A critical issue in workflow management is the assignment of tasks and cases to the appropriate *agent* (also called *workflow participant*), in order to execute activities or to supervise their execution. The approach adopted by most WfMSs consists in allowing the definition of an *organization schema* that describes the structure of the organization relevant to workflow management. In the organization schema, agents are grouped in several ways, according for instance to their skills or to the organizational unit they belong to.

In the definition of the workflow schema, processes and activities are (statically) bound to elements of the organization schema (e.g, to *roles* or *organizational units*) rather than to individual agents. This approach decouples the definition of the process from the definition of the agents, and provides more flexibility, since changes in the organization schema do not affect process definitions.

At run-time, as a task is scheduled for execution, the WfMS determines all the agents allowed to execute it, and inserts the task into their worklists. As an agent *pulls* the task from his/her worklist in order to start working on it, the task is removed from the worklists of the other agents. For simplicity, we assume to have an organization model based on roles and agents; the extension to more complex organization models is straightforward.

In the following sections, we introduce the authorization model for the assignment of agents to roles and of tasks to roles, and a rule-based paradigm for its specification and enforcement.

3 An advanced role-based authorization model for workflows

In this section, we present an advanced role-based access control model for the assignment of tasks to agents for workflow execution. A basic role-based authorization model [31] is commonly adopted in most commercial and research workflow systems, to comply with organizational security policies. In fact, these policies are often expressed in terms of roles

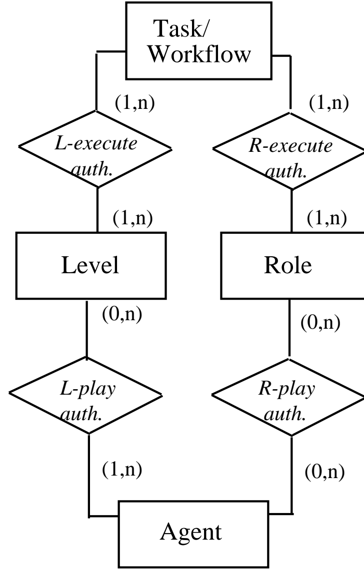

Figure 2: The proposed workflow authorization model

rather than in terms of specific individuals. Roles are job functions describing the authority and responsibility conferred to the individuals that are assigned to them. According to this basic model, authorization to execute tasks are associated with roles, and agents are authorized to play one or more roles. We extend this basic model by introducing: i) the concept of *organizational level*, and ii) *authorization constraints*.

In the following sections, we first describe roles, organizational levels, agents and authorizations for the assignment of tasks to authorized agents. Then, we describe the extension of the model with authorization constraints.

3.1 Elements of the model

Elements of the authorization model are shown in Fig. 2, using the Entity-Relationship notation. The model considers agents and tasks, roles and organizational levels (from now on called level for simplicity), and authorizations.

Agents and tasks

Agents and tasks are the active entities of the model [20], causing the data to flow among tasks or change the workflow system state.

- An *agent*, denotes a processing entity of the organization to which tasks can be assigned for execution. An agent can be a human user or an application. We denote by A the set of agents in the workflow system.
- A *task*, corresponds to an execution unit of a workflow. We denote by T the set of

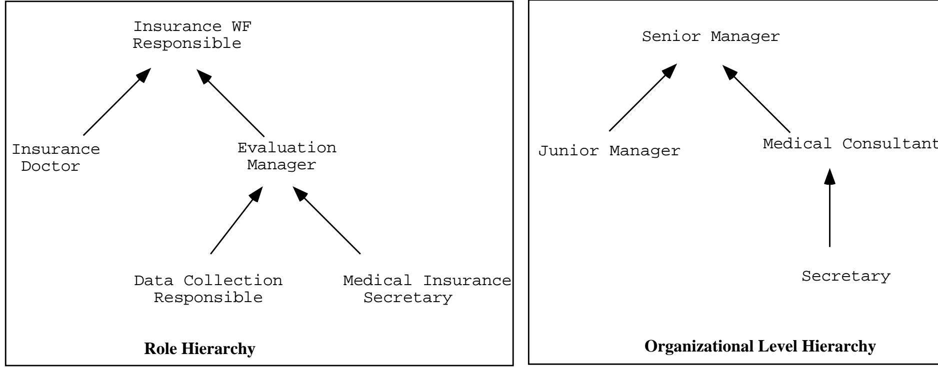

Figure 3: An example of role and level hierarchy for the workflow of Fig.1

tasks in the workflow system.

Roles and levels

Roles and levels are the organizational elements of the model, describing capabilities of agents to execute tasks according to organizational assets.

- A *role* represents a job function. We denote by R the set of roles defined in the workflow system.
- A *level* represents a functional level of the organization according to the organization chart. We denote by L the set of levels defined in the workflow system.

As in conventional role-based authorization models [31], roles and levels are organized in hierarchies. The *role hierarchy* $RH \subseteq R \times R$ is a partial order relation on R, denoted by $\leq_R$, also called "role dominance" relation. Given $r, r' \in R$, $r \leq_R r'$ holds if r precedes r' in the order, graphically represented by an edge going from r' to r in the hierarchy. An example of role hierarchy for the `Medical Insurance` workflow of Fig. 1, is shown in Fig. 3. With reference to the figure, we have that `Evaluation Manager`$\leq_R$ `Data Collection Responsible`. The role hierarchy relation implies inheritance of authorizations between roles. If $r \leq_R r'$, authorizations specified for r' (e.g., `Data Collection Responsible`) are inherited by r (e.g., `Evaluation Manager`). Analogously, a *level hierarchy*, $LH \subseteq L \times L$, is defined for levels which is a partial order relation $\leq_L$ on L. Given $l, l' \in L$, $l \leq_L l'$ holds if l precedes l' in the order. The graphical notation for the level hierarchy follows the same conventions of the role hierarchy. The availability of a hierarchical model also allows the definition of authorization rules based on X.509 certificates or LDAP directories, which have a hierarchical structure analogous to the one presented in this paper. However, support for digital certificates is not included in our prototype implementation.

Authorizations

Authorizations define privileges for task assignment and execution. A task can be assigned only to agents that are authorized based on the role and level they have in the organization. To regulate task assignment, in the model we introduce the two kinds of authorizations:

- *Play authorizations*
- *Execute authorizations*

Play authorizations determine the roles (*R-play* authorizations) and the levels (*L-play* authorizations) to which agents have to be assigned. We denote by RPA and LPA the set of *R-play* and *L-play* authorizations, respectively, and by $PA = RPA \cup LPA$ the whole set of play authorizations defined in the workflow system.

Definition 1 (R-play authorization) An R-play authorization $rpa \in RPA$ is a triple:

$$\langle a, \texttt{play}, r \rangle$$

where $a \in A$ and $r \in R$, stating that agent a is assigned to role r.

For example, authorization ⟨ `John,play,Evaluation Manager` ⟩ specifies that `John` is assigned to the `Evaluation Manager` role.

Definition 2 (L-play authorization) An L-play authorization $lpa \in LPA$ is a triple:

$$\langle a, \texttt{play}, l \rangle$$

where $a \in A$ and $l \in L$, stating that agent a is assigned to level l.

For example, authorization ⟨`Mary,play,Secretary`⟩ specifies that `Mary` is assigned to the `Secretary` level.

Execute authorizations determine the tasks that roles (*R-execute* authorizations) and levels (*L-execute* authorizations) can execute in a workflow. We denote by REA and LEA the set of *R-execute* and *L-execute* authorizations, respectively, and by $EA = REA \cup LEA$ the whole set of execute authorizations defined for in the workflow system.

Definition 3 (R-execute authorization) An R-execute authorization $rea \in REA$ is a triple:

$$\langle r, \texttt{execute}, t \rangle$$

where $r \in R$ and $t \in T$.

An authorization $\langle r, \texttt{execute}, t \rangle$ states that role r is authorized to execute task t. For example, authorization ⟨Evaluation Manager,execute,Evaluation⟩ is a permission for role Evaluation Manager to execute the Evaluation task.

Definition 4 (L-execute authorization) An L-execute authorization $lea \in LEA$ is a triple:

$$\langle l, \texttt{execute}, t \rangle$$

where $l \in L$ and $t \in T$.

An authorization $\langle l, \texttt{execute}, t \rangle$ states that level l is authorized to execute task t. For example, authorization ⟨Secretary,execute,Issuing⟩ is a permission for role Secretary to execute the Issuing task.

Following conventional role-based authorization models, inheritance of execute authorizations occurs in role and level hierarchies, respectively. Authorizations inherited by a certain role/level are dynamically derived using the following derivation rules:

(R1) Rule 1. For any $t \in T$, $op = execute$ if $r \leq_R r'$ then $\langle r', \texttt{execute}, t \rangle \rightarrow \langle r, \texttt{execute}, t \rangle$

(R2) Rule 2. For any $t \in T$, $op = execute$ if $l \leq_L l'$ then $\langle l', \texttt{execute}, t \rangle \rightarrow \langle l, \texttt{execute}, t \rangle$

Consider the *Medical Insurance* workflow depicted in Fig. 1 and the role and level hierarchies of Fig. 3. If authorization ⟨Evaluation Manager, execute, Evaluation⟩ is defined for role Evaluation Manager, then authorization ⟨ Insurance WF Responsible, execute, Evaluation ⟩ is derived for role Insurance WF Responsible (rule R1). Analogously, if authorization ⟨Junior Manager, execute, Evaluation⟩ is defined for level Junior Manager, then authorization ⟨Senior Manager, execute, Evaluation⟩ is derived for role Senior Manager (rule R2).

3.2 Authorization constraints

Authorizations in $PA \cup EA$ constitute a "basic" set of authorizations capable of satisfying some of the requirements of security policies of an organization. In particular, associating authorizations with roles and levels has the advantage of reducing the complexity of authorization management, since the number of roles is smaller than the number of agents and it is not necessary to update authorizations when agents change position and duties in the organization. Authorizations in $PA \cup EA$ are also of static nature, that is, once they are defined, they apply to all instances of a workflow schema, in any instant of time, and independent of the activity execution history. Consequently, they are not flexible enough to

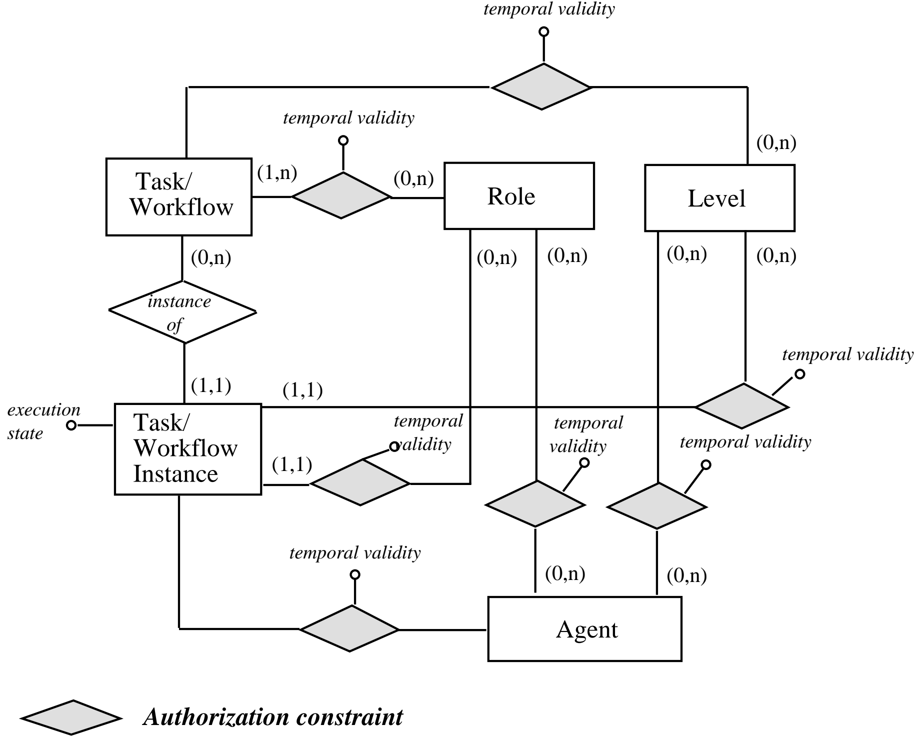

Figure 4: The advanced workflow authorization model with authorization constraints

capture all possible security policy requirements of an organization on workflow execution, which require the capability of expressing and enforcing constraints. Examples of constraints that can be required on a workflow are the following: "two different roles/agents must execute two tasks T_1 and T_2 in a given workflow WF" (separation of duties); "a role R can execute a task T in a given workflow WF only for a given period of time" (restricted task execution); "at the least K roles must be associated with a workflow WF in order to start its execution" (cooperation); "a case is inhibited to all agents for a given period of time" (inhibition). Constraints require the capability of specifying authorizations that are of dynamic nature, either because they are bound only to a specific workflow instance (e.g., the inhibition constraint above) or because they are time-dependent (e.g., the restricted task execution constraint) or because they are related to the activity execution status (e.g., the separation of duties constraint). To specify authorizations of dynamic nature and make the model more advanced and compliant with organizational policies, we introduce authorization constraints.

The advanced authorization model is shown in Fig. 4, where authorization constraints are graphically represented as grey relationships. By means of authorization constraints, it is possible to specify instance, temporal, and history authorizations.

Instance authorization constraints. Authorizations in $PA \cup EA$ are defined at the work-

flow/task schema level, and are valid for all corresponding instances. An instance authorization constraint expresses an authorization that has validity only for specific workflow/task instances. To model instance authorizations, an entity named `Task/Workflow instance` is defined in the model.

Temporal authorization constraints. Authorizations in $PA \cup EA$ are always valid, independent of the time they are considered. A temporal authorization constraint expresses an authorization that has a validity limited in time. To model temporal authorizations, an attribute `temporal validity` is specified for authorization constraints.

History authorization constraints. Authorizations in $PA \cup EA$ do not take into account information on past execution of activities in a workflow. An history authorization constraint expresses an authorization which depends on the status of the system at a certain point in time, during the activity flow execution (e.g., separation of duties for tasks). To take into account history authorizations, an attribute `execution state` is added to task/workflow instances, keeping track of the state of the instance in the system (i.e., running or completed).

We denote by AC the set of authorization constraints defined for in the workflow system. Consequently, the set of authorizations that are defined in the system is given by $PA \cup EA \cup AC$. For the advanced model the close system assumption holds, that is, each task assignment operation is rejected unless an authorization for it is defined in $PA \cup EA \cup AC$. A security manager, or workflow administrator, is responsible for granting and revoking authorizations in the system.

In the next section, we describe how to implement authorizations in $PA \cup EA \cup AC$ using the active database technology.

4 Active rule support to authorization constraint management

The previous section has outlined the main characteristics of authorization constraints for workflows. Some of them result in demanding requirements for an authorization constraint language and for the system implementation enforcing them. These requirements suggest the adoption of a rule-based approach to the definition and enforcement of authorization constraints. In fact, authorizations should be granted or revoked at the occurrence of specific events (e.g., specific points in time, modifications to system or workflow relevant data, activations of tasks or cases), and as certain conditions over system or workflow relevant data are verified (e.g., when workflow relevant datum has a given, critical value). Thus, a

constraint may be described by an ECA rule, where:

- the *event* part defines when new authorizations/prohibitions may need to be enforced (in addition to the ones of the reference model);
- the *condition* part verifies that the occurred event actually requires the modification of authorizations, and determines the involved agents, roles, tasks, and cases;
- finally, the *action* part enforces authorizations and prohibitions.

This section will show how ECA rules may be exploited for defining and enforcing authorization constraints.

ECA rules can also support derivation of authorizations. The advanced authorization model is based on role and level hierarchies, where authorizations defined for a given role (respectively, level) are propagated to (i.e., inherited by) its parents. Consequently, a derivation mechanism is necessary to derive propagated authorizations along the hierarchies. ECA rules are a very well suited paradigm for managing authorization derivation, and we will show how they can be adopted to implement the derivation mechanism.

An additional motivation that suggests the use of active rules as a modeling and implementation paradigm is that many commercial WfMSs (such as Changengine by Hewlett-Packard [22] and MQ Workflow by IBM [25]), and several research prototypes (such as WIDE [20] and TriGSflow [26]) execute on top of an active database, that offers rule definition and execution support.

In the following, we first present the structure of the authorization base on the top of which active rules are executed. Then, we introduce a few assumptions and a notation for active rules and we discuss the use of active rules for defining and enforcing authorization constraints and for managing derivation of authorizations. Finally, we show how the WfMS may determine the set of agents authorized to execute a given task instance.

4.1 Schema of the authorization base

Figure 5 shows the schema of the authorization base, in terms of relations. In the figure, underlined attributes for each relation denote the primary key. For ease of presentation, we assume that names of tasks, levels, and roles are unique within the WfMS.

Relations *RoleHierarchy* and *LevelHierarchy* define the hierarchies of roles and levels, by specifying which are the parents of each role or level. The hierarchies are shared by all workflows. Note that the pair $\langle Role, Parent\rangle$ is the primary key, meaning that we allow multiple parents for each role or level. These relations are populated as the organization schema is defined; relation *RoleHierarchy* is also typically updated as a new workflow schema

RoleHierarchy (Role, Parent)
LevelHierarchy (Level, Parent)
R-Play (Agent, Role)
L-Play (Agent, Level)
R-Execute (TaskName, Role, Type)
L-Execute (TaskName, Level, Type)
Force (TaskName, Case, Agent)
Revoke (TaskName, Case, Agent)

Figure 5: Relational schema of the workflow authorization base

is defined, since often new roles need to be introduced for the newly specified schema. For instance, in our sample workflow system that includes the `Medical Insurance` workflow of Fig. 1, the authorization base relations are populated as follows:

RoleHierarchy	Role	Parent
	Data Collection Responsible	Evaluation Manager
	Medical Insurance Secretary	Evaluation Manager
	Evaluation Manager	Insurance WF responsible
	Insurance Doctor	Insurance WF responsible

LevelHierarchy	Level	Parent
	Secretary	Junior Manager
	Junior Manager	Senior Manager
	Medical Consultant	Senior Manager

Relations *R-Play* and *L-Play* define play authorizations. These relations are populated as the organization schema is defined; relation *R-Play* is also typically updated as a new workflow schema is defined. In our sample workflow system the above relations are populated as follows:

R-Play	Agent	Role
	Judy	Evaluation Manager
	John	Insurance WF responsible
	...	...

L-Play	Agent	Level
	Brenda	Secretary
	Mary	Secretary
	Judy	Junior Manager
	...	...

Due to the close system assumption, we only need to store authorizations, and the absence of an authorization corresponds to a prohibition.

Relations *R-Execute* and *L-Execute* defines execute authorizations. They are populated as a new workflow schema is defined, and can be later modified at the occurrence of specified events on time or workflow data. Attribute *Type* defines whether the authorization is explicitly specified for the role/level or if it has been derived by means of derivation rules. In our sample workflow system, these relations are populated as follows[1]:

R-Execute	TaskName	Role	Type
	Data collection	Data collection Responsible	explicit
	Evaluation	Evaluation Manager	explicit
	Decision	Evaluation Manager	explicit
	Medical examination	Insurance Doctor	explicit
	Customer dossier preparation	Medical Insurance Secretary	explicit
	Notification of rejection	Medical Insurance Secretary	explicit
	Issuing	Medical Insurance Secretary	explicit
	Filing	Medical Insurance Secretary	explicit

[1]For ease of presentation, we only show explicitly assigned authorizations; derived authorizations, also stored in tables *R-Execute* and *L-Execute*, can be computed on the basis of explicit ones, as shown in Section 4.4.

L-Execute	TaskName	Level	Type
	Data collection	Secretary	explicit
	Evaluation	Junior Manager	explicit
	Decision	Junior Manager	explicit
	Medical examination	Medical Consultant	explicit
	Customer dossier preparation	Secretary	explicit
	Notification of rejection	Secretary	explicit
	Issuing	Secretary	explicit
	Filing	Secretary	explicit

Finally, relations *Force* and *Revoke* allow the definition of *instance-dependent* authorizations or prohibitions. Relation *Force* specifies that the next execution of a given task within a given case must be assigned to the agent specified in the relation. Note that the pair $\langle task, case\rangle$ is the primary key (see Figure 5), meaning that the assignment to a task of a given case can only be forced to one agent. Relation *Revoke* defines prohibitions for an agent to execute a given task of a given case. For this relation, the primary key is formed by all the three attributes, meaning that the authorization to execute a task of a given case may be revoked to several agents. The choice of explicitly storing case-specific prohibitions, despite the close system assumption, is motivated by the observation that case-specific prohibitions are usually a small set, and thus it is surely more convenient to store prohibitions rather than storing the complementary set, i.e., explicitly defining all authorizations for every task in every case, which would be a very large set.

Many other relations are necessary to operate a workflow in an actual implementation. In particular, the WfMS database will include information about tasks, cases, and agents. In the following, we assume the existence of a *TaskInstance* table, that stores generic attributes related to task executions:

TaskInstance(TaskInstanceId,TaskName,Case,Executor)

This table stores one tuple for every task instantiation, including the task identifier, the name, the case to which the task belongs, and the executing agent. Note that we are not concerned with the exact schema of *TaskInstance* and of other support relations: analogous authorization rules can be written, depending on the actual database schema describing the workflow structures and enactment. The *TaskInstance* relation we propose here has only the purpose of demonstrating the feasibility of our approach.

4.2 Assumptions and notation for active rules

We need very few assumptions about active rules, which therefore apply to most products and research prototypes of active databases [28, 36]. For the ease of specification, in this paper we use the relational model and high-level, Datalog-like active rules. An analogous approach can be devised for object-oriented databases and object-oriented rule languages, and indeed in Section 5 we will show an object-oriented implementation of our approach within the WIDE WfMS.

We assume that rules follow the ECA paradigm, that is, they are triggered by specific events, include a declarative condition and a sequence of procedural actions.

In the following, we use a simple notation for active rules, borrowed from [3]. Rules have the following syntax:

rule ⟨ rule-name ⟩
    when ⟨ events ⟩
    if ⟨ condition ⟩
    then ⟨ action ⟩

- The *event* part contains a list of events, which are restricted to instant or periodic temporal events, insertions, deletions, and qualified updates (on specific attributes).

- The *condition* part contains a boolean expression of predicates. Predicates can be either simple comparisons between terms (variables or constants), or database predicates with the usual *Datalog* [35] interpretation; the special predicates *inserted*, *updated*, and *deleted* unifies with newly inserted and deleted tuples.

- The *action* part contains a sequence of commands. Bindings are passed from conditions to actions by means of shared variables (denoted by capital letters): variables in the action are bound to the values that are computed for them by the condition. Bindings are not changed by the actions' evaluation, which consists of a sequence of set-oriented primitives.

Several examples of active rules will be provided in the remainder of the section.

A rule is triggered when one of its associated events occurs. Rule processing is started whenever a rule is triggered, and consists in the iteration of rule selection and rule execution.

- Rule selection may be influenced by static, explicitly assigned priorities: whenever two or more rules are triggered, the one with the maximum priority is executed (ordering among rules with the same priority is non-deterministic). When a rule is selected, it is also detriggered.

- Rule execution consists of evaluating the condition and next executing the action only if the condition is true. Action execution may cause events and therefore trigger other rules.

We assume that rule processing initiates immediately after events are detected, and terminates when all triggered rules have been executed. We assume that the rate at which events are generated is lower than the actual rate at which events are processed, thus avoiding backlogs.

4.3 ECA rules for authorization constraint enforcement

Earlier in this section we have motivated the use of active rules as a suitable paradigm for modeling and enforcing authorization constraints. In the following we provide some examples of such rules. The examples show triggers belonging to the different categories introduced in Section 3.2, and are defined for the sample `Medical Insurance` workflow schema shown in Figure 1.

Example 1: Binding of duties.
This rule enforces the *BindingOfDuties* constraint for agents between tasks `Data_Collection` and `Issuing`. This constraint is triggered as an agent pulls a task in order to execute it, since in that instant all the *BindingOfDuties* constraints involving the pulled task become "defined", i.e., it is known who the executing agent is. We assume that pulling a task corresponds, from a database perspective, to the update of attribute *Executor* in a tuple of relation *TaskInstance*. The value of attribute *Executor* is initially set to NULL, until the task is pulled by an agent for execution.

In the condition part, the rule identifies the executor of the `Data_Collection` task and the involved case and task instance, while the action part inserts a tuple in the database that specifies that task `Issuing` must be assigned to the agent who executed task `Data_Collection` in the same case. The definition of the rule is as follows:

rule	`bindingOfDuties`
when	`updated(TaskInstance.Executor)`
if	`updated[TaskInstance(-, "Data_Collection",CASE,AGENT)],`
then	`insert[Force("Issuing",CASE,AGENT)]`

Example 2: Separation of duties.
This rule enforces the *SeparationOfDuties* constraint between tasks `Evaluation` and `Decision`. It is triggered as a task is pulled by an agent: once it is known who the executor of a task

`Evaluation` is, the rule can revoke to him/her the permission to execute task `Decision` in the same case. The semantics of the condition part is analogous to that of the previous rule.

```
rule   separationOfDuties
when   updated(TaskInstance.Executor)
if     updated[TaskInstance(-,"Evaluation",CASE,AGENT)],
then   insert[Revoke("Decision",CASE,AGENT)]
```

Example 3: Restricted task execution.
This example shows a rule (actually a pair of rules) enforcing the time-dependent constraint *RestrictedTaskExecution*. The first rule grants the permission to all agents playing role `Secretary` to execute task `Filing` every day at 8 a.m. The second rule revokes the same permission every day at 6 p.m. Note that, unlike the previous example, these rules define a case-independent constraint, which holds for every instance of the `Medical Insurance` schema.

```
rule   temporalGrant
when   @8:00
if     true
then   insert[R-Execute("Filing","Secretary","explicit")]

rule   temporalRevoke
when   @18:00
if     true
then   delete[R-Execute("Filing","Secretary","explicit")]
```

4.4 ECA rules for authorization management

One of the key concepts of our approach is the notion of *derived authorization* in the role/level hierarchies, defined by rules R1 and R2 of Section 3.1. If the workflow administrator authorizes a role/level to execute a given task T, then a *derived* authorization to execute T is implicitly defined for its ancestors in the role or level hierarchy. For instance, an execute authorization granted to role `Insurance Doctor` for a given task causes the same authorization to be derived for role `Insurance WF responsible`.

This section defines a set of active rules, operating on the authorization base defined above, that specifies the operational semantics of the derivation mechanism. Rules are activated as explicit authorizations are granted or revoked (corresponding, from a database

perspective, to insertions into and deletions from the authorization tables, respectively), and compute derived authorizations.

We first present rules that manage insertions of new execute authorizations for roles and levels, and then we describe rules managing deletions.

```
rule deriveRoleAuthorization
    when  inserted(R-Execute)
    if    inserted[R-Execute(TASK,ROLE,-)],
          RoleHierarchy(ROLE,PARENT)
          not (R-Execute(TASK,PARENT,"derived"))
    then  insert[R-Execute(TASK,PARENT,"derived")]

rule deriveLevelAuthorization
    when  inserted(L-Execute)
    if    inserted[L-Execute(TASK,LEVEL,-)],
          LevelHierarchy(LEVEL,PARENT)
          not (L-Execute(TASK,PARENT,"derived"))
    then  insert[L-Execute(TASK,PARENT,"derived")]
```

The rules are analogous for roles and levels. In the following, we describe how rule `deriveLevelAuthorization` operates (rule `deriveRoleAuthorization` operates analogously): as a new (explicit or derived) authorization is granted to a level (event `inserted(L-Execute)`), the condition part determines the parent levels for which the authorization has been defined (predicate `inserted[L-Execute(TASK,LEVEL,-)], LevelHierarchy(LEVEL,PARENT)`), and verifies that a derived authorization has not already been granted for the same task (predicate `not (L-Execute(TASK, PARENT,"derived"))`), in order to propagate it to the parents. If the condition is verified, the action part is executed, inserting the (derived) authorization into the *L-Execute* table (action `insert[L-Execute(TASK,PARENT,"derived")]`).

Note that the condition (and the primary key of relations R-Execute and L-Execute) ensures that only one explicit and one derived authorization are defined for a given $\langle task, role \rangle$ or $\langle task, level \rangle$ pair, in order to avoid redundancy. Note also that the action part re-triggers the same rule, thereby iterating the derivation process. Triggering terminates either when all parents of the newly authorized role or level are themselves already authorized to execute the task, or when the derived authorization is granted to the root of the hierarchy.

We now describe rules *deleteRoleAuthorization* and *deleteLevelAuthorization*, that manage deletions of derived authorizations.

rule deleteRoleAuthorization
  when deleted(R-Execute)
  if deleted[R-Execute(TASK,ROLE,-)],
    RoleHierarchy(ROLE,PARENT),
    not (RoleHierarchy(SIBLING,PARENT),
    R-Execute(TASK,SIBLING,-))
  then delete[R-Execute(TASK,PARENT,"derived")]

rule deleteLevelAuthorization
  when deleted(L-Execute)
  if deleted[L-Execute(TASK,LEVEL,-)],
    LevelHierarchy(LEVEL,PARENT),
    not (LevelHierarchy(SIBLING,PARENT),
    L-Execute(TASK,SIBLING,-))
  then delete[L-Execute(TASK,PARENT,"derived")]

The semantics of rule *deleteLevelAuthorization* are as follows (the semantics for rule *deleteRoleAuthorization* is analogous): the rule is triggered by a revocation of an explicit or derived authorization related to a given $\langle TASK, LEVEL \rangle$ pair. The condition part determines whether derived authorizations to execute $TASK$ should be revoked to its parents as well. A derived authorization is revoked to a given level only if none of its descendants holds an authorization to execute the same task. Thus, the condition verifies that no immediate descendant has the authorization[2] (`not (LevelHierarchy(SIBLING,PARENT), L-Execute(TASK,SIBLING,-))`), and, if so, the action part revokes the authorization to the parents.

Note that if an explicit authorization was granted to $PARENT$, it still holds, that is, the parent is still authorized to execute the task. The motivation that led to the adoption of these semantics is explained by the following example: suppose that an execute authorization is explicitly granted to level `Secretary`, and an execute authorization for the same task is also explicitly granted to level `Junior Manager`: the adopted semantics guarantees that if we revoke the authorizations to level `Secretary`, the junior manager still has the authorization to execute the task.

Thus, derived authorizations are computed in advance, and are stored in the authorization base. Although this implies the need for a larger storage space, it provides two significant advantages that suggest the adoption of this approach:

[2]There is no need of checking all the descendants: the derivation mechanism ensures that if a given level or role does not have a derived authorization, then none of its descendants will.

1. reduced computational overhead: derivations have to be computed when explicit authorizations are modified rather than each time is task is activated. We expect modifications of authorizations to be significantly less frequent than task activations.

2. reduced delay between task scheduling and assignment: all authorizations are pre-computed and a simple query determines the set of agents allowed to execute the task.

4.5 Assigning tasks to agents

The previous sections have detailed the workflow authorization model and presented how active rules can be used for defining the operational semantics of the model and as an implementation mechanism. Active rules appropriately modify the authorization tables (*R-Play*, *R-Execute*, *L-Play*, *L-Execute*, *Revoke* and *Force*), so that the WfMS can always determine the set of agents authorized to execute a task of a given case.

We first observe that the authorization base can be divided into a case-independent part and a case specific part.

The case-independent part is composed of relations *R-Play*, *R-Execute*, *L-Play*, and *L-Execute*. These relations enable the maintenance of a case-independent view, **Case-indipExecutor**(TaskName, Agent), that defines which are the agents authorized to execute a given task. The view is defined as follows:

```
Case-indipExecutor (TASKNAME,AGENT):- R-Play(AGENT,ROLE),
                                      R-Execute(TASK,ROLE,-),
                                      L-Play(AGENT,LEVEL),
                                      L-Execute(TASK,LEVEL,-)
```

Since the view is case-independent, it requires a reduced storage space. Therefore, for the same motivations stated above, it should be materialized, to achieve better system performance at the cost of a (relatively small) increase in the storage space. View materialization can also be performed by means of active rules. The derivation of active rules for materialized view maintenance has been discussed in several papers (see, e.g., [15]).

We are now ready to present how the set of agents authorized to execute a task is determined. Consistently with the approach presented in this paper, the semantics will be defined in terms of active rules.

We assume that the activation of a new task corresponds, from a database perspective, to the insertion of a new tuple in the *TaskInstance* table. As the new tuple is inserted, the authorized agents are determined; a set of active rules, triggered by the insertion in

the *TaskInstance* table, examines the contents of tables *Force* and *Revoke*, as well as the content of view *Case-indipExecutor*, and fills in table **Authorized**(TaskInstanceId,Agent), that defines the agents authorized to execute the newly activated instance[3]. Insertions in the *Authorized* table are managed by two rules: *authorizeForcedAgents* and *authorizeAgents*. Rule *authorizeForcedAgents* checks whether the task must be assigned to a specific agent, due for instance to a binding of duties authorization constraint. The rule, in the condition part, checks that an entry in the *Force* table exists for the task in the case under consideration, and that the authorization has not been revoked. If the condition holds, then the action part inserts a tuple in the *Authorized* table, in order to notify that the agent involved is authorized to execute the task for that case. Rule *authorizeForcedAgents* is defined below:

```
rule    authorizeForcedAgents
when    inserted(TaskInstance)
if      inserted[TaskInstance(TASKID,TASKNAME,CASE,-)],
        Force(TASKNAME,CASE,AGENT),
        not(Revoked(TASKNAME,CASE,AGENT))
        Case-indipExecutor(TASKNAME,AGENT)
then    insert[Authorized(TASKID,AGENT)]
```

As detailed in the definition of table *Force*, only one agent can be forced as the executor of a task, therefore only one tuple is inserted. Note also that it may be possible that the authorization, possibly due to erroneous or conflicting authorization constraints, has been revoked, or even that it was never granted (this is checked by predicate `not(Revoked(TASKNAME,CASE,AGENT),Case-indipExecutor (TASKNAME,AGENT))`). In this case, no entry is made in the *Authorized* table. This paper is not concerned with system policies that defines the appropriate behavior when no authorized agent exists. This is an exceptional situation, and we expect it to be managed by the workflow engine for instance by sending a message to the workflow responsible asking him/her to suggest the appropriate executing agent.

Next, we present rule *authorizeAgents*, that determines the authorized agents in case no agent has been forced. The rule first consider case independent authorizations (predicate `Case-indipExecutor(TASKNAME,AGENT)`), and then restricts the set to those agents to whom the permission has not been revoked for the case under consideration (predicate `not (Revoked(TASKNAME,CASE,AGENT))`). Finally, the condition checks that no forced agent has been defined for the task in this specific case (predicate `not(Force(TASKNAME,CASE,-))`). If

[3]We assume that the task instance identifier *TaskInstanceId* is unique within a given WfMS domain, otherwise the case identifier must be included among the attributes of this relation

the condition holds, then the action part inserts one or more tuples in the *Authorized* table, defining the agents authorized to execute the newly activated instance.

```
rule    authorizeAgents
when    inserted(TaskInstance)
if      inserted[TaskInstance(TASKID,TASKNAME,CASE,-)],
        Case-indipExecutor(TASKNAME,AGENT)
        not (Revoked(TASKNAME,CASE,AGENT)),
        not(Force(TASKNAME,CASE,-))
then    insert[Authorized(TASKID,AGENT)]
```

Once the set of authorized agents has been determined, the WfMS must select the agents to which the task is assigned for execution (i.e., inserted in the corresponding worklists). A possible approach consists in assigning the task to all authorized agents. While this solution is feasible and does not violate any authorization constraint, it is inpractical, since the worklists of high-level employees would be quickly filled-up. An alternative and more reasonable policy consists in first trying to assign tasks to authorized agents placed in the lower positions of the role/level hierarchies. If none of these agents is available, then tasks are assigning to agents placed in higher positions. Many other policies are possible, such as assigning tasks depending on the agents' workload, following a "round-robin" approach, or selecting a "dispatching agent" that will assign the task to a suitable agent. The definition of these policies is outside the scope of this paper, since they are not concerned with authorization constraints, provided that agents are selected among the set of authorized ones. For a discussion on task assignment policies the reader is referred to [20].

Finally, we observe that rules may be also adopted in order to handle exceptional situations related to task assignment to agents. Examples of exceptional situations are modifications of authorizations to execute a given task occurring after the task has been scheduled, or even when it is already in execution. Suitable rules can detect the exceptional event and possibly notify to the WfMS that the task needs to be reassigned to a different agent. Examples of such exception handling rules are provided in [7, 12].

5 Implementation in the WIDE workflow management system

This section shows a concrete implementation of authorization constraints and derivation rules within the WIDE WfMS. We first give a brief description of the WIDE rule language

Chimera-Exc and of the rule execution engine. Then, we present the WIDE authorization base and the Chimera-Exc rules defined for authorization constraint management; finally we show how derivation triggers are implemented.

5.1 The Chimera-Exc language

The WIDE workflow management system provides a process model that, in addition to the traditional modeling constructs for specifying workflow schemas, allows the definition of *triggers* (also called rules in the following). WIDE triggers conform to the ECA paradigm and are specified in *Chimera-Exc* [8, 10], a language derived from the object-oriented database language *Chimera* [14]. Triggers are a fundamental complement to the graph-based representation of a business process (typical of most conceptual workflow models, and of WIDE), since they allow the workflow administrator to model and react to several types of synchronous and asynchronous events that may occur in a workflow execution. Chimera-Exc triggers react to the following types of events:

- *Data events*, which correspond to updates to system or workflow relevant data. They may correspond to a constraint violation, to a task or case cancellation, or to the unavailability of an agent.
- *External events*, that are user defined, application-specific, (such as a document arrival, a telephone call, an incoming e-mail), and are explicitly raised by external applications.
- *Workflow events*, raised as a case or task is started or completed. They are expressed through predefined events such as *caseStart, caseEnd, taskStart(taskName), taskEnd(taskName)*.
- *Temporal events*, expressed as deadlines, time elapsed since a certain instant, or cyclic periods of time.

Each rule in Chimera-Exc can monitor multiple events, with a disjunctive semantics: the rule is triggered if any of its triggering events occurs.

The *condition* part of a rule is a predicate on system and workflow relevant data at the time of the condition evaluation, which indicates whether the event must be managed. Chimera-Exc requires that the object-oriented schema upon which rules execute is logically defined. Chimera-Exc rules access three types of classes: WIDE classes, workflow -specific classes, and event handling classes.

- *WIDE classes* include description of agents, roles, tasks, and cases. These classes are workflow-independent, and are predefined in the system; objects are created when new

roles, agents, tasks or cases are created. For instance, Chimera-Exc rules may refer to WIDE attributes concerning tasks by accessing the attributes of the class `task`, which has a workflow-independent structure. An object of this class is created whenever a new task is instantiated (independently on the workflow schema it belongs to); hence, the executor of a case can be accessed by defining a variable `T` ranging upon the `task` class, and then by accessing `T.executor`. For instance, `task(T), agent(A), T.executor=A, A.name="Judy"` selects the tasks whose executor is Judy.

- *Workflow-specific classes* store workflow relevant data. Each case will be represented as an object within this class, created when the case is started.
- *Event handling classes* store information carried by occurred events. For instance, the `externalEvent` class is referred in order to access the parameters of an occurred external event.

A condition is a query that inspects the contents of the WIDE database. Queries consists of a formula evaluated against the state of the database and of a list of variables, to which a set of bindings is assigned by the evaluation of the formula. Conditions include class formulas (for declaring variables ranging over the current extent of a class, e.g., `medicalInsurance(C)`; `C` in this case ranges upon object identifiers of the `medicalInsurance` class), type formulas (for introducing variables of a given type, e.g., `integer(I)`), and comparison formulas, which use binary comparison between expressions (e.g., `T.executor="Mark"`). Terms in the expressions are attribute terms (e.g., `C.examNeeded`) or constants. The predicate `occurred`, followed by an event specification, binds a variable defined on a given class to object identifiers of that class which were affected by the event. For instance, in `agent(A), occurred(create(agent),A)`, `A` is bound to an object of the agent class that has been created. If the result of a query is empty (i.e., no bindings are produced), then the condition is not satisfied and the action part is not executed. Otherwise, bindings resulting from the formula evaluation are passed to the action part in order to perform the reaction over the appropriate objects.

The *action* (or reaction) part may consist of calls to the WfMS, requiring a particular service, or of manipulation of workflow data (i.e., object creation, modification, or deletion). Calls to the WfMS may request to activate or terminate cases or tasks, to reassign a task to a different agent, or to send notification messages to workflow agents.

Rules in WIDE are defined either in the context of a specific workflow or as global. In the first case, we say that rules are *targeted* to a workflow, and their side effects are propagated only to the cases and tasks of that workflow. Global (untargeted) triggers may

be used for generic authorization policies, shared by all schemas, while targeted triggers define authorization constraints for a given schema.

5.2 Rule execution

Chimera-Exc rules are executed by a system called FAR (Foro Active Rules), integrated with the WIDE engine, called FORO. The FAR system executes Chimera-Exc triggers in *detached* mode, i.e., in a separate transaction with respect to the triggering one. This choice has several motivations: Chimera-Exc actions are outside the database context, and cannot be rolled back. Therefore, we have to be sure that the triggering event actually occurred before processing it, which means that the triggering transaction must commit. Furthermore, the use of the detached mode allows us to manage data events in an uniform way with respect to the other event types, and provides considerable advantages in the performance, allowing the processing of several events generated by different transactions with the execution of a single rule. The semantic and performance issues that advised the use of a detached execution mode are discussed in detail in [8]. This feature is relevant to authorization management since it influences our approach towards the development of WIDE derivation rules, as detailed later in this section.

The FAR system is engineered in order to be easily portable to different database platforms. This is a fundamental requirement for a commercial WfMS, that typically needs to be able to execute within heterogeneous environments. When the FAR runs on top of relational DBMSs, then the authorization base (along with the entire WIDE database) is stored in relational tables. A suitable *SQL2Chimera* application translates the relational definitions into Chimera classes. The relational DBMS is then accessed through a layer that maps Chimera conditions into SQL queries, thereby enabling the portability of WIDE onto different (relational) platforms. This is also the case of the current implementation, which runs on top of the Oracle database server, version 7.3. FAR also makes a very limited use of database specific features, so that it can be ported to other relational or object-oriented platforms with a reduced effort.

5.3 The WIDE authorization base

We extended the set of WIDE classes in order to enable the implementation of the authorization mechanism defined in Section 4. The WIDE prototype already included classes *agent* and *role*, whose objects store basic information about agents and roles, such as the agents' names, location, vacation period, and roles' names and descriptions. We added a class *level*, analogous to class *role*, and we also added classes that enable the implementation of role

and level hierarchies and the definition of play and execute authorizations. They are the object-oriented counterpart of the relations defined in Section 4, and their schema is shown in Figure 6.

5.4 Authorization constraint enforcement in WIDE

Although WIDE triggers were initially conceived for modeling exceptional situations that may occur during process execution [8], the capability of capturing and reacting to several types of events and the rich expressive power of the condition and action language make them an appropriate construct for modeling and managing authorization constraints. Since the expressive power of Chimera-Exc includes that of the Datalog-like rules defined in Section 4.2, and given that the authorization base of WIDE has the same structure of the one defined in Section 4.1, a possible implementation consists in mapping each rule of Section 4.3 with a Chimera-Exc rule that implements the same semantics: relational operations (insert, update, delete) are mapped into their corresponding Chimera-Exc object-oriented operation (i.e., create, modify, delete), and Datalog conditions are replaced by Chimera-Exc predicates. For instance, rule `bindingOfDuties` of Section 4.3 can be implemented in Chimera-Exc by means of the following trigger:

define trigger	`bindingOfDuties`
events	`modify(task.executor)`
condition	`task(T), occurred(modify(task.executor), T),`
	`T.name="Data_Collection"`
actions	`create(force[(taskName:"Issuing",caseId=T.caseId,`
	`agentId=T.executor], F)`

However, WIDE users are not allowed to define rules that directly (i.e., within the rule's action) update the content of system data, since erroneously defined rules could harm the integrity of the system. Thus, rules notify to the workflow engine the actions to be performed, and the engine then performs these actions as and when appropriate. For instance, while rules are in principle capable of reassigning tasks to different agents, when a task needs to be reassigned a rule sends a notification message to the workflow engine, that in turn selects the new agents (according to whatever assignment policy has been defined, of which rules may be unaware), dispatches the task to the appropriate agents, and updates the database accordingly. Therefore, consistently with this approach, rules modeling and managing authorization constraints do not directly modify the authorization base, but rather notify to the engine the new authorizations or prohibitions.

```
define object class roleHierarchy
            attributes roleId: role, parentId:role
            constraints key (R:roleId, P:parentId)
end

define object class levelHierarchy
           attributes  levelId: level, parentId:level
           constraints key (L:levelId, P:parentId)
end

define object class r-Play
            attributes agentId: agent, roleId: role
            constraints key (A:agentId, R:roleId)
end

define object class l-Play
           attributes  agentId: agent, levelId: level
           constraints key (A:agentId, L:levelId)
end

define object class r-Execute
            attributes taskName: string, roleId: role, type: string
            constraints key (T:taskName, R:roleId, P:type)
end

define object class l-Execute
            attributes taskName: string, levelId: level, type: string
            constraints key (T:taskName, L:levelId, P:type)
end

define object class force
           attributes taskName: string, caseId: case, agentId: agent
           constraints key (T:taskName, C:caseId)
end

define object class revoke
           attributes taskName: string, caseId: case,  agentId: agent
           constraints key (T:taskName, C:caseId, A:agentId)
end
```

In order to enable the definition of triggers modeling and enforcing authorization constraints, we have extended Chimera-Exc actions by adding several WfMS calls: *r-Play*, *r-Execute*, *l-Play*, *l-Execute*, *revoke*, and *force*. Primitives *r-Play* and *l-Play* enable the modification of play authorizations for roles and levels. They both have three parameters: the role/level, the agent involved in the authorization, and a binary value that defines whether the case-independent authorization is to be granted or revoked (allowed values are `grant` or `revoke`). Similarly, *r-Execute* and *l-Execute* enable the definition of execute authorizations, and also have three parameters: the role/level, the task involved in the authorization, and a parameter that defines whether the case-independent authorization is to be granted or revoked.

Primitives *revoke* and *force* enable the modification of case-specific authorizations. The *revoke* primitive revokes to an agent the authorization to execute a given task in a given case, while *force* constrain the WfMS to assign a task of a given case to the specified agent. They both have three parameters: the name of the agent, the name of the task, and the case identifier. For instance, referring to the workflow schema of Figure 1, the call `revoke("Judy","Issuing",medicalInsurance305)` revokes to Judy the permission to execute task `Issuing` in case 305 of the Medical Insurance process.

We next present examples of Chimera-Exc triggers enforcing the authorization constraints presented in Section 4.3. **Example 1: Binding of duties**.

define trigger	`bindingOfDuties` **for** `medicalInsurance`
events	`modify(Task.executor)`
condition	`task(T), occurred(modify(task.executor), T),`
	`T.name="Data_Collection"`
actions	`force(T.executor,"Issuing",T.caseId)`

Example 2: Separation of duties.

define trigger	`separationOfDuties` **for** `medicalInsurance`
events	`modify(Task.executor)`
condition	`task(T), occurred(modify(task.executor), T),`
	`T.name="Evaluation"`
actions	`revoke(T.executor,"Decision",T.caseId)`

Example 3: Restricted task execution.

```
define trigger   temporalGrant for medicalInsurance
events           8/hours during days
condition        true
actions          r-Execute("Secretary","Filing",grant)


define trigger   temporalRevoke for medicalInsurance
events           18/hours during days
condition        true
actions          r-Execute("Secretary","Filing",revoke)
```

5.5 Derivation triggers in WIDE

Unlike triggers defining and enforcing authorization constraints, derivation triggers are global and predefined. They are not visible to the user, and their purpose is to implement the derivation mechanism. For this reason, we are allowed to define rules that directly modify the database state. Furthermore, in order to enforce the semantics defined in Section 4, derivation triggers need to be executed immediately as an explicit authorization is granted or revoked, within the context of the same transaction, otherwise another transaction could see a database state in which an authorization is defined for a role/level but not for their ancestors.

Chimera-Exc rules are executed periodically, and are detached. Therefore, they are not suited for the implementation of derivation rules. In WIDE we use instead native, immediate Oracle triggers, which are defined in the WIDE database as the system is installed. The implementation of WIDE onto different database platforms will require the modification of these triggers; however, we only exploit basic active functionality, so that similar triggers are implementable, with minor changes, on most commercial database systems, consistently with the requirements and objectives of the WIDE project [20].

The derivation triggers defined in Oracle are analogous to the those defined in Section 4.4. In the following, as an example, we show how rule *deriveRoleAuthorization* of Section 4.4 is implemented as an Oracle trigger. With respect to the corresponding Datalog-like rule, the Oracle trigger does not check previous existence of the tuple it is inserting (if a tuple already exists, the insertion is rejected), and the value `derived` in the *type* attribute is implicitly added as default value, so there is no need to explicitly define it in the action part of the rule.

```
CREATE TRIGGER deriveRoleAuthorization
       AFTER INSERT ON r-Execute
```

```
FOR EACH ROW
INSERT INTO r-Execute (taskName, roleId)
select distinct r-Execute.taskName, roleHierarchy.parentId
from r-Execute, roleHierarchy
where r-Execute.taskName=:new.taskName AND
roleHierarchy.roleId=:new.roleId
```

6 Related work

In this section, we compare our work with respect to related work on workflow authorization constraints and with respect to triggers and security mechanisms in commercial workflow management systems.

Workflow authorization constraints. A logic-based language for the specification and verification of authorization constraints in workflow systems is described in [5]. Here, different types of constraints are introduced for workflows, based on an role-based access control model. Static, dynamic, and hybrid constraints are identified for consistency analysis purposes. Static constraints can be evaluated before workflow execution. Dynamic constraints can be evaluated only during workflow execution. Hybrid constraints are a combination of the two and can be partially evaluated without executing the workflow. In our approach, we employ triggers for both specification and enforcement of authorization constraints. Due to the trigger-based mechanism, a constraint is evaluated run-time, during the execution of a workflow. However, triggers can be statically analyzed to evaluate confluence and termination properties, using techniques analogous to the ones proposed in the active database field. We have discussed issues related to authorization constraint analysis in [6].

In our approach, authorization constraints also model time and history-dependent authorizations. Access control models have been recently proposed specifically for workflows. In [1], a workflow authorization model is described defined in way that the authorization flow is synchronized with the activity flow. The model is based on the concept of "authorization template" associated with each workflow task, to grant authorizations to a task only when the task starts, and revoke them when it terminates. Temporal authorizations are defined that have a validity only within the expected duration of a certain task. Analogously to this model, in our approach, it is possible to specify authorization constraints allowing agents to access workflow data only from within an authorized task.

More general aspects related to security and integrity requirements for business processes implemented as interorganizational workflows are discussed in [21]. In [17], discretionary

and mandatory access controls, object-oriented security, and the Clark and Wilson model are revisited for evaluating their applicability to collaborative workflows. To better cope with workflow requirements, capability of specifying and enforcing authorization constraints is required, to be able to specify several organizational security policies on task execution and assignment.

Research work relevant to this paper also includes task-based authorization models and separation of duties in computerized systems, both defined in the context of distributed applications. With task-based authorizations [30], authorizations are seen in terms of tasks rather than individual subject and objects. The concept of "authorization-task" is introduced as a unit to manage the authorizations in distributed applications, which can be refined into authorization-subtasks. The separation of duties constraint in computerized system has been introduced in [29], where transactional control expressions have to enforce computerized controls analogous the ones in manual, paper-based systems.

Authorization constraints in commercial workflow management systems. Although workflow management systems have become very popular in recent years, and hundreds of commercial products presently exist on the market, only recently the workflow community has started to address the problem of providing flexible authorization mechanisms, also pushed by the need for increased security imposed by cross-organizational interactions and by the use of workflows for supporting e-commerce transactions. Most WfMSs only provide basic functionality for the definition of workflow authorizations: authorization models typically allow the definition of roles and levels, and enable the definition of play and execute authorizations which hold for all workflow instances. Furthermore, roles and levels are often not structured in a hierarchy, thereby making authorization management more complex.

However, a few products allow the definition of limited forms of instance-, history-, and time-dependent authorizations. For instance, IBM's *MQ Workflow* [25] allows the definition of the *binding of duties* constraint: the executor of a task can be restricted to be the same executor of another task in the same case or to be the case initiator. *Staffware2000* [32] also enables the definition of the binding of duties constraint, although this must be statically defined, it holds for all instances, and cannot be defined in tasks that joins flows from multiple tasks. In addition, Staffware also allows the definition of authorizations that are valid only for a specified time period. *InConcert* [27], by InConcert Inc., in addition to static binding of agents to tasks and of tasks to roles, allows the definition of external applications, that are invoked at task assignment time to determine the role to which the task should be assigned. *COSA* [2], by Baan, is the commercial WfMS that provides the greatest flexibility in defining authorization constraints and task assignment criteria. COSA allows the definition of agent groups and group hierarchies, analogous to the role/level hierarchies presented in this paper,

where authorizations can be inherited along the hierarchies. With respect to authorization constraints, COSA provides a simple language that enables the definition of the binding and separation of duties constraints and of time-dependent authorizations. Changengine, by Hewlett-Packard, is the commercial product with the richest and most flexible resource model [23]. Task assignments are specified by a *resource rule*, executed each time a task is scheduled by the system. The rule, written in a Changengine-specific language, may invoke one or more methods on several business objects which encode the logic for agent selection. Such business objects, called *resource agents*, may for instance query a database or an LDAP directory in order to select the appropriate agent.

The above approaches have several limitations with respect to the approach presented in this paper: in fact, they are less powerful in the class of time-, instance-, and history-dependent constraints they can model, allowing only the definition of a few types of constraints. Furthermore, they do not allow the definition of constraints that depend on the state of several workflow instances, and are incapable of managing *global* constraints, i.e., constraints applied to every task or case (for instance, a business policy may require that no agent executes the same task for more than twice, regardless of the specific task or workflow). Global constraints can be instead modeled in WIDE with untargeted triggers. Finally, external applications or agent expressions, defined in order to determine the set of authorized agents, have to be computed each time a task is activated, rather than each time an authorization is modified, with performance disadvantages.

Active rules and workflow enactment. Rules have been used as a mechanism to *enact* workflows in a few research prototypes such as [9, 19, 26]. In fact, WfMSs basically are event-driven engines, that react to task completions by selecting the tasks to be activated next. Hence, in principle, the ECA paradigm is suitable for describing enactments; however, in practice, active rules are not used for workflow enactment, due to a rather low performance of the resulting workflow engine implementation; a drawback of the use of active rules for workflow enactment is that they must build the current context of the workflow by inspecting the database state, while a workflow interpreter typically maintains information about the current context within easy-to-access state variables. However, active rules can be used for defining a precise, albeit operational, semantics of workflow enactment.

A few commercial products, such as *COSA* and *InConcert*, provide event-action triggers as a workflow *modeling* construct. However, triggers can be only exploited as a complement to a workflow specification, in order to define how the execution flow should be modified when specified events occur and are captured by the trigger. None of these systems provides a trigger mechanism capable of defining and enforcing authorization constraints.

7 Concluding remarks and future work

Authorization constraint definition and management in workflow processes should be faced since the early phases of the workflow design. In this paper, we have illustrated an advanced role-based model for authorization management in workflows. In our approach, authorization constraints are defined and enforced by means of active rules, executed on top of a suitable authorization base. We have shown that active rules are a powerful and flexible modeling formalism for the definition of authorization constraints, and that they can be also adopted for their enforcement . Furthermore, active rules can be also exploited in order to manage derivations of authorizations. The integration of workflow, active database, and security technologies is a characteristic issue of this paper; in our opinion and experience, this integration allows for flexible reaction to changes during workflow enactment.

The advantage of the proposed approach is that rules, besides being a convenient conceptual means at authorization specification time, suggest also the implementation strategy for authorizations. We are experimenting such advantage in the WIDE workflow system, which relies on an active database where authorization constraints are defined as triggers in the Chimera-Exc language. We believe that the proposed approach is of general interest and is suited for being adopted by commercial systems, which indeed execute on top of active database platforms.

The use of triggers for authorization constraints has also the advantage of exploiting *authorization patterns* for their specification. Authorization patterns are abstract definitions of triggers that can be reused when designing a new workflow process by instantiating parts of the pattern to meet the authorization requirements of the new process. A basic set of authorization patterns has been presented in [12]. Authorization patterns are stored in a catalog, together with other patterns developed in WIDE for exception modeling and handling [7]. Facilities for the workflow administrator to define, add, and delete authorizations to/from the authorization base are under development in the framework of the *GLAD* tool, originally developed for global authorization management in federated databases [11], and now being extended to workflow authorization management. The user interface is an interesting issue to be experimented in authorization management in cooperative work environments, as well as the interaction between the workflow and the external applications and databases. In fact, an open issue of our research concerns how to handle authorizations for access to application data of a workflow process. Such issue, briefly tackled in [12], as well as the issue of accessing distributed federated databases [11], needs further investigation in both the authorization definition process and in the implementation aspects due to conceptual design problems and to system performance issues. The interaction with external information systems has been tackled in [4] and is another open issue of our research.

acknowledgements The authors are thankful to Dr. Angelica Tescaro for investigating some ideas presented in this paper during her degree Thesis. This work has been partially supported by the Italian Consortium for Informatics (CINI), by the Italian National research Council in the framework of "Progetto Strategico Informatica nella Pubblica Amministrazione - DEMOSTENE Project", and by the Esprit Project WIDE.

References

[1] V. Atluri and W. Huang. An extended petri-net model for supporting workflows in a multilevel secure environment. In *Proceedings of the 10th IFIP TC11/WG11.3 International Conference on Database Security*, Como, Italy, Sept. 1996. Chapman & Hall.

[2] Baan Company N.V. - COSA Soultions. *COSA Reference Manual*, 1998.

[3] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago, Chile, Sept. 1994.

[4] L. Baresi, F. Casati, S. Castano, M. Fugini, I. Mirbel, and B. Pernici. The WIDE workflow design methodology. In *Proceedings of WACC'99*, San Francisco, California, USA, 1999.

[5] E. Bertino, E. Ferrari, and V. Atluri. A flexible model supporting the specification and enforcement of role-based authorizations in workflow management systems. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, Santiago, Chile, Nov. 1997.

[6] F. Casati, S. Castano, and M. Fugini. Enforcing workflow authorization constraints using triggers. *Journal of Computer Security*, 6(4), 1999.

[7] F. Casati, S. Castano, M. Fugini, I. Mirbel, and B. Pernici. Using patterns to design rules in workflows. Technical Report 97.065, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1997.

[8] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. Technical Report 98.081, Dipartimento di Elettronica e Informazione, Politecnico di Milano, July 1998.

[9] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Deriving active rules for workflow enactment. In *Proceedings of the International Conference on Database and Expert Sys-*

tems Administration(DEXA'96), Lecture Notes in Computer Science, Springer Verlag, Zurich, Switzerland, Sept. 1996. Springer-Verlag, Berlin.

[10] F. Casati, M. Fugini, and I. Mirbel. An Environment for Designing Exceptions in Workflows. In *Proceedings of the 10th International Conference on Advanced Information Systems Engineering (CAiSE'98)*, Lecture Notes in Computer Science, Springer Verlag, Pisa, Italy, June 1998. Springer-Verlag, Berlin.

[11] S. Castano, S. De Capitani Di Vimercati, and M. Fugini. Automated derivation of global authorizations for database federations. *Journal of Computer Security, IOS Press*, 5(4), 1997.

[12] S. Castano and M. Fugini. Rules and patterns for security in workflow systems. In *Proceedings of the 12th IFIP TC11/WG11.3 International Conference on Database Security*, Chalchidiki, Grece, July 1998. Kluwer Academic Publishers.

[13] S. Ceri and R. Ramakrishnan. Rules in database systems. *ACM Comput. Surv.*, 28(1):109–111, Mar. 1996.

[14] S. Ceri and R. Ramakrishnan. Rules in database systems. *ACM Computing Surveys*, 28(1):109–111, Mar. 1996.

[15] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*, Lecture Notes in Computer Science, Springer Verlag, pages 577–589, Barcelona, Spain, Aug. 1991. Springer-Verlag, Berlin.

[16] R. Cochrane, H. Pirahesh, and N. Mendonça Mattos. Integrating triggers and declarative constraints in SQL database sytems. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*, Bombay, India, Sept. 1996. Morgan-Kaufmann.

[17] E. Ellmer, G. Pernul, and G. Quirchmayr. Security for Workflow Management. In *Proceedings of 6th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, Washington D.C., Oct. 1994.

[18] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, Apr. 1995.

[19] A. Geppert and D. Tombros. Event-based distributed workflow execution with EVE. Technical Report 96.05, University of Zurich, 1995.

[20] P. Grefen, B. Pernici, and G. Sanchez. *Database Support for Workflow Management: the WIDE Project*. Kluwer Academic Publishers, 1999.

[21] G. Herrmann and G. Pernul. A general framework for security and integrity in interorganizational workflows. In *Proceedings of 10th International Bled Electronic Commerce Conference*, 1997.

[22] Hewlett Packard. *Changengine Process Design Guide*, 2000.

[23] Hewlett Packard. *Changengine Resource Management Guide*, 2000.

[24] D. Hollingsworth. The workflow reference model. Technical Report WFMC-TC-1003, 1.1, Workflow Management Coalition, 1995.

[25] IBM. *MQ Series Workflow - Concepts and Architectures*, 1998.

[26] G. Kappel, P.Lang, S. Rausch-Schott, and W. Retschitzegger. Workflow management based on objects, rules, and roles. *IEEE Data Engineering*, 18(1):11–18, Mar. 1995.

[27] D. McCarthy and S. Sarin. Workflow and transactions in InConcert. *IEEE Data Engineering*, 16(2):53–56, June 1993.

[28] N. W. Paton, O. Diaz, M. H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N. W. Paton and M. H. Williams, editors, *Proceedings of First Workshop on Rules in Database Systems*, WICS, Edinburgh, Scotland, Aug. 1993. Springer-Verlag, Berlin.

[29] R. Sandhu. Separation of duties in computerized information systems. In S. Jajodia and C. Landwehr, editors, *Database Security IV: Status and Prospects*. North-Holland, 1991.

[30] R. Sandhu. Task-based authorization: A paradigm for flexible and adaptable access control in distributed applications. In *Proceedings of 16th NIST-NCSC National Computer Security Conference*, Baltimore, MD, USA, 1993.

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

[32] Staffware Corporation. *Staffware2000 White Paper*, 1998. Available at http://www.staffware.com/home/products/Staffware2000WP.zip.

[33] The Workflow Management Coalition. Process definition interchange v 1.1. Technical Report WfMC-TC-1016-P, Workflow Management Coalition, 1999.

[34] The Workflow Management Coalition. Terminology and Glossary. Technical Report WfMC-TC-1011, 3.0, Workflow Management Coalition, 1999.

[35] J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989. 2 Volumes.

[36] J. Widom and S. Ceri. *Active Database Systems.* Morgan-Kaufmann, San Mateo, California, Aug. 1996.