User Modeling and Control in Adele System

Walcelio L. Melo * Noureddine Belkhatir Jacky Estublier

Laboratorie de Genie Informatique, BP 53X, 38041 Grenoble, FRANCE
{wmelo, belkhatir, jacky}@imag.imag.fr

Published in the *Proc. of the 4th Int'l Conf. on Computing and Information*, Toronto, Ontario, Canada, May 28-30,1992

1 Introduction

Development and maintenance of large software involves teams where each person has specific responsibilities and capacities in the *software process*. Software engineering environments (SEE) should provide ways of controlling human responsibilities and capacities. These characteristics are often modeled in terms of *roles*. People may play different roles at different times, and a given role may be played by different people at different times. In a SEE, several persons play different *roles* and perform different *tasks* to achieve specifics *goals*.

An example of this sort of SEE includes roles such as a **project manager**, responsible for coordinating a software development team, a **configuration manager**, responsible for CM procedures and policies, **software engineers**, responsible for developing and maintaining the software product, **testers**, responsible for product validation.

This sample shows that roles may be structured, and customized in sub-roles. For instance, a software development group includes a project manager, a configuration manager, a set of SE, and a tester. These aspects of human behavior should be modeled and monitored by a SEE in order to provide real support for the software processes.

In the remainder of this article, we show how Adele provides both dynamic and static views of user role in a SEE. In order to achieve this objective, this article is organized as follows: Section 2 presents how human characteristics are controlled by traditional systems and why they are not sufficient to support the role as presented before. Section 3 is a quick overview of the Adele system. Section 3.2 discusses and justify capacities included in Adele to model and control roles. The article concludes with a summary of the status of work.

*Melo is supported by Technological and Scientific Development National Council of Brazil (CNPq)

2 User modeling in traditional systems

In all SEEs, we find access control mechanisms. At very least, a file system distinguishes between the owner of a file and other users; Unix provides the notion of group. Commercial DBMSs provide features to define access rights (e.g. read, write) for different database items. We can classify this kind of control in the following categories:

- there is an **access list** associated with each object. For each object, this list indicates who can perform actions on it. For efficiency reasons, this mechanism is often very simple and the access list is reduced to fixed size information. For example, Unix has only three action classes (read, write and execute) and three user classes (owner, group and others).
- there is an **rights list** associated with each user. For each user, this list indicates the objects on which actions can be performed. This is not usual in most systems.

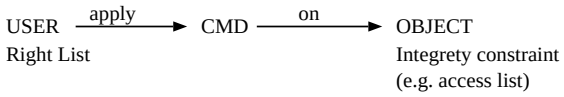

Figure 1: Rights list versus access list

We believe that both approaches have advantages and drawbacks, and that they can be combined. For

example, a user U can drive a car C if U has a driver's license (it is U's right to drive cars), AND if car C can be driven by U (U is in the access list of C: U is the owner or the owner's friend, etc.). Thus both must be verified.

The majority of SEEs, such as DSEE, EPOS[?], MARVEL[?] etc, do not manage user roles explicitly. A user is known by his login name. This approach cannot be used to model hierarchical groups and role based access control.

A few systems, such as TRIAD[?] and ARCHIPEL AGENDA[?], take these problems into account.

In TRIAD, user roles are explicitly modeled in a conceptual modeling language (CML). Each role type has attached its available actions (e.g. create/delete objects, create other roles, etc.) and its database views. User group facilities are also available.

In ARCHIPEL AGENDA, users, roles and tasks are modeled. When a user is allocated a role to carry out a task, a work context is created. A user environment comprises all his Work Contexts and contains the tools and objects on which he has rights. A user may delegate his tasks to another user, provided the latter has similar capabilities (rights).

3 Adele background

Adele is a SEE designed to provide support for programming-in-the-many activities, such as support for team, configuration and versions. Adele is built around a database based on an entity-relationship model extended with version and object-oriented concepts (multiple inheritance and encapsulation). Adele SEE integrates **a product model** used to describe the structure of the software (source, code, documents, etc), an **activity model** used to take into account the behavior aspects of the SEE. It is based on an event-condition-action formalism (ECA) [?], and a **process model** designed to describe users, tools, software processes, etc.

This paper concentrates mainly on the third aspect, showing how user resources (user roles) can be managed by a process-oriented SEE. We are interested in the structuring and monitoring of user activities during the software process and the description of users as part of the conceptual software product model. This type of model must take into account the multiple roles involved during the software process and allow the dynamic role changing.

3.1 Modeling user roles

The static aspect of users are described using the object-oriented (OO) paradigm, i.e., each user is an instance of a user type. Object-oriented concepts, such inheritance (ISA), are used to specialize user roles. User behavior is described in terms of which actions he can perform on which objects.

In the next section, we give an overview of Adele's trigger mechanism, then we describe the use of this mechanism for the management of user rights.

3.1.1 Adele's activity manager

The activity manager is the active component of the Adele database. It is used as a general purpose rule formalism to maintain database integrity, to integrate external tools in the Adele environment and to synchronize parallel software activities. This manager is based on a trigger mechanism making it possible to execute actions in the database and to communicate with external tools. The user can define object behavior as well as propagations along relationships.

Adele triggers take the following form:

```
ON event DO Action;
```

Where `event` is a predicate over the system state, object state and the current activities (query, navigation as well as changes) occurring on objects.

```
EVENT delete = [command = rm];
```

An `Action` is a program in the Adele Language. An Adele language instruction can be a logical expression, an Adele command or a Unix command. This language is a simple imperative language, tailored to access Data Base information and navigate easily through arbitrary relationships. It is a meta substitution language (late binding of parameters and variables) that looks like the Unix shell, except that variables are multi-valued attributes, with provision for complex query and set operators. Three built-in actions are provided:

1. `raise X -e E` raises the event E on object `X`.
2. `ABORT` causes the current action to be undone.
3. `ACCEPT` continues the current action.

3.2 Modeling user groups in Adele

In the following section, we present a user role modeling strategy using inheritance, rules and trigger mechanisms. The customization of user type is achieved by object type inheritance, right checking by triggers.

3.2.1 Using triggers and inheritance

In the first solution users are modeled in the same way as standards objects.

```
TYPEOBJECT object;
 PRE ON true DO
   raise !username -e right_control;

TYPEOBJECT  soft_eng IS user;
 PRE ON right_control DO
  IF [ state = official ] AND
     [ level > 3 ]
  THEN ABORT;
END soft_eng;

TYPEOBJECT administrator IS user; ...

DEFEVENT
 mod_state=[!cmd=mda,!optvala=state];

TYPEOBJECT program IS object;
 PRE ON mod_state DO
  IF [state = official] AND
     [username.type != administrator]
  THEN ABORT ;
```

Using the trigger mechanism, we can describe the database objects that a user may see and what operations can be applied (see [?] for more details about Adele's trigger). "`User`" a predefined type, is the root for all subsequent user refinement. In the same way as for all other Adele object types, user type may be refined and specialized in other user types.

This example shows the definition of the `administrator` sub-type and `soft_eng` sub-type. Since each Adele command has a security level, (called `level`), and all modification commands have a level superior to 3, instances of soft_eng cannot execute modification commands on objects in the "`official`" state.

Conversely, program objects have an access list: only the administrator is able to modify the state (attribute) for programs in "`official`" state (before the command).

This solution can be implemented without any change to the standard Trigger mechanism and matches our requirement: both access lists and rights lists coexist. We have large facilities for defining rights control. However this solution has severe drawbacks:

- A given user can only be of one type, he cannot change roles dynamically.
- Modeling multiple roles can be achieved by multiple inheritance, but standard inheritance for triggers produces an intersection of role, not union. However using only ACCEPT, allows for union of privilege. In practice defining multiple roles in this way is tricky.
- There is a confusion between triggers executed when a user is an object, and triggers to execute as the actor of a command (its rights list).
- In practice privileges can only be defined in terms of intention, since the same definition is shared by all instances. If rights like "IF [object = xyz] THEN ..." are used, all users of the same type will refer explicitly to the same object instance.
- performance. The explicit definition of rights, using pre-rules to express the object domain in terms of intention and managed by a trigger mechanism, has a strong negative impact on performance. The first evaluations carried out on the prototype have shown severe performance degradation.

These difficulties led us to define rights and users in a different way.

3.2.2 Multiple user definition

Let us suppose a single user U can be instantiated an arbitrary number of times in different user types. Each definition of U defines its characteristics and privilege for a different role.

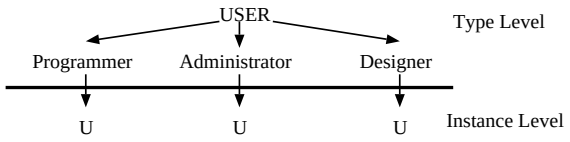

Figure 2: User definition graph

The basic type `user` has a specific trigger bloc called `RIGHT` executed each time a user of that type tries to execute a command; it replaces logically the `raise` primitive in the previous solution.

The command `select_role` permits a user to move from one role to another. The mixture of role type with role selection provides several advantages. First, the type lattice is a compact and concise description of a user-role organization, including compound roles built by multiple inheritance. Secondly, the multiple instantiation of the same user in different roles is a natural way to express and sequentialize the different roles a user can play in the organization.

User instances Each user is specific; his rights are almost never exactly the same as the default user. As a programmer he may have rights on some documents (those he wrote), even if no documenter role has been assigned to him. Conversely he may have rights only on some programs, not on all programs in the database. Each individual user instance must be customized. We allow each user instance to own a RIGHT trigger bloc. For example, for user U, as programmer.

```
MANUAL john ;
 RIGHT
  IF [ name = mydoc ] THEN ACCEPT ;
  IF NOT ([name = foo] OR [name = bar])
     THEN ABORT ;
```

User U has extended rights on document mydoc, but rights are restricted to use only objects called `foo` and `bar`. This mechanism is in complete opposition with the OO paradigm; all instances of a class are potentially different, they act in some way as a type. Inheritance between the instance and its type uses the same algorithm as inheritance between types.

Sub Databases It is possible, but not practical, to specify explicitly in the user type the list of object instances he has rights on, because it can be a large list, and because this list changes frequently. We complemented our rights mechanism by a sub database mechanism: a user is always into a sub-database, and can only access the objects of its current database, he can change dynamically of sub-database. In this way, rights can be expressed only in terms of intention.

4 evaluation

We found user modeling very demanding; users are not standard objects (they may even cheat!). We presented user model roles as an application based on both Adele trigger and Object-Oriented concepts of the data model without any change. Several extensions were needed. First, each individual user can be customized, in extension or restriction; second, the role a user plays can be changed dynamically, and a hardwired `RIGHT` trigger bloc ensures efficient control. Sub Database filtering complements our mechanism.

From team management point of view, we emphasize the concepts for expressing user role models. It allows to describe and structure any team organization. From technology point of view, we have shown how Object-Oriented concepts coupled with ECA rules and trigger allows to describe these role models.

We are currently experimenting the model by developing an environment which supports the management of a large team.

The concurent use of different aspects allows a separation of concern. Capabilities shared by a set or users are modeled by a user sub_type (a role); the specifics of each user are described as part of the user instance. Rules specific to a type of instance are described in the access list (triggers) associated with this type of object.

We have shown that, with O.O. and triggers, few extensions are needed to implement an extremely powerfully (may be too much?) and fairly efficient (may be not enough?) user modeling and access control.