A Component-Based Access Control Monitor

Zhiming Liu, Charles Morisset, and Volker Stolz

United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau SAR, China
{lzm,morisset,vs}@iist.unu.edu

Abstract. A control of access to information is increasingly becoming necessary as the systems managing this information is more and more open and available through non secure networks. Integrating an access control monitor within a large system is a complex task, since it has to be an "all or nothing" integration. The least error or mistake could lead to jeopardize the whole system. We present a formal specification of an access control monitor using the calculus of refinement of component and object systems (rCOS). We illustrate this implementation with the well known Role Based Access Control (RBAC) policy and we show how to integrate it within a larger system. Keywords: Component, Access Control, RBAC, Composition

1 Introduction

In the design of most information systems, security issues are usually considered as a secondary task. Roughly speaking, the first objective is to design a "working" system and then, if it is possible within the time and budget constraints, try to inject some security mechanisms. As a result of this approach, these mechanisms are often poorly designed and their lack of integration in the global system can cause major flaws. A classical example is when one can find, on a public server, both an SSH server (quite secure) and a Telnet server (usually non secure). This is often due to the fact that the telnet server is installed first, while the server is not public yet and the SSH server is only installed afterwards. The consequence is that not only the server is subject to simple attacks due to the weakness of Telnet, but the server administrator thinks he is protected because he has installed an SSH server and so will not enforce other security mechanisms.

Of course, there exist some domains where security issues are highly considered, usually because human beings or important sums of money are at stake. This is particularly true for the military field, where many security mechanisms have been created, including cryptographic techniques and access control models. In these fields, the question of security is addressed from the beginning, as a part of the global system, which usually ensures a greater confidence in the system.

The component-based approach in rCOS allows us to integrate the security aspect closer into the software engineering process instead of trying to add it later. Some works already combine UML modeling and security issues, like UMLsec [15]. However, the latter focuses more on the cryptographic problematics while we try to address access control. Basically, the access control problematics consists in defining a policy, that is the set of granted (or denied) accesses to a system by subjects (users, processes, etc) over some objects (files, resources, processes, etc). An access control monitor (or reference monitor) is a program enforcing the policy, that filters all the accesses to grant only the ones allowed and to deny the ones forbidden.

The concept of access control can be split into the *interface* to the access control monitor and its actual *implementation*. As the rCOS software development methodology of component systems uses the familiar features of UML for the development process, we expect that a formalisation of monitoring will make this important aspect more accessible to practitioners and students.

But not only allows this to talk about security in familiar terms to software engineers, it also eliminates the disconnect between the software and reasoning about security aspects that might introduce additional errors when both were handled in different formalisms.

By using the notion of rCOS component interfaces, we can make sure that the monitor component is at least used syntactically correct and according to its protocol, that is the sequence of possible traces. Even the implementation of a monitor profits from being modeled in rCOS: we can give a high-level, mathematical specification and use the rCOS refinement techniques to obtain an executable implementation [4,19]. Additionally, the available formal methods for rCOS, like the verification of component composition through process algebra and model checking, allows reasoning about the correctness and properties of the composition of components realising access control with components providing the system behaviour that is to be protected.

From a normative point view, according to the Common Criteria [2], which is one of the authoritative references in the domain of safety and security systems, a reference monitor is an abstract machine which enforces the access control policies in a system and it should have the three following properties:

- unsafe subjects cannot interfere with it,
- unsafe subjects cannot circumvent its controls,
- it is simple enough to be analyzed and its behaviour understood.

We present here an approach addressing these properties. The main contributions of this paper are:

- the implementation of an access control model based on a recent formalization, which clearly separates the notion of policy and the one of implementation of a policy, and introduces some new concepts, such as the semantics of requests,
- another case study of the rCOS tool, showing how its features can be successfully applied to the access control problematics.

We first introduce in section 2 a formalisation of access control policies and models, with the example of the Role Based Access Control (RBAC) policy.

This formalisation, together with the different proofs, ensures that the monitor behaviour is understood and could be analyzed. It also guarantees that unsafe subjects cannot interfere with it, at least on the design level. Indeed, our approach, as most of classical software engineering approaches, does not ensure that the implementation will not be modified at run-time, or that the hardware the system is running on is safe. Then, in section 3, we outline the main features of rCOS and section 4 gives a description of a specification of the access control monitor within rCOS. Thanks to the component-based approach, it is possible to ensure that the monitor cannot be circumvented by hiding the non secure interfaces and integrating the monitor directly into the system. Finally, we present in this section a way to integrate such a monitor in a larger system.

2 Defining Access Control Models

In this section, we present a way to define access control systems, based on two main concepts: policy and model. Due to space limitation, we only give here an overview, a complete definition can be found in [14,21]. The *policy* is the description of the system on which it is enforced, defined as a state machine together with the notion of secure states. Hence, an access control policy is considered here as a functional property that a state machine must satisfy. Of course, the definition of the policy also includes all the information relevant to the definition of the system, such as subjects, objects, security information, etc. At this point, a policy can be seen as "static", since it is expressed over states, and a state is a snapshot of the system. We then introduce the notion of the *model*, which is basically a policy together with a set of requests (and, as we will see later, the semantics of these requests). These requests are a way for subjects to access objects. Lastly, it is possible to define an implementation (or several) for a model, through a transition function and a set of initial states. Intuitively, this implementation corresponds to a reference monitor and should be proved correct with respect to the security policy, that is, returning a secure state for any secure state and any request.

2.1 Access Control Policy

We first define the main entities of a system: $\mathcal{S}$ is the set of subjects (active entities initiating actions in the system), $\mathcal{O}$ is the set of objects (passive entities on which actions are made) and $\mathcal{A}$ is the set of access modes (read, write, append, etc). In this paper, we represent an access by a triple (s, o, a) expressing that a subject s accesses an object o according to the access mode a. Hence, we define the set of accesses $\mathbb{A}$ as the Cartesian product $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$. Other approaches are possible to represent accesses. For example, in order to deal with "joint access" of a group of subjects over an object, as in [20], $\mathbb{A}$ can be defined as $(\wp(\mathcal{S}) \backslash \{\emptyset\}) \times \mathcal{O} \times \wp(\mathcal{A})$.

Then we can define an access control policy $\mathbb{P}[\rho] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma, \Omega)$, where ρ is the security parameter (that is all the information needed to define the policy),

$\mathcal{S}$ the set of subjects, $\mathcal{O}$ the set of objects, $\mathcal{A}$ the set of access modes, Σ the set of states and Ω the security predicate, characterizing the secure states.

Role-Based Access Control models are a set of fairly new models first introduced in the nineties. The key concept of theses models is the notion of a role, which can be seen as an abstraction of the one of subject. Intuitively, a subject can have many roles and an object can be accessed by many roles. This indirection eases the management of subjects within a system, since the authorizations are related to roles (which are supposed not to change a lot) rather than subjects (who can change a lot). We give here a version of RBAC based on the RBAC92 model [7], in order to simplify the presentation, but some more complex versions can be found, as RBAC96 [24] extends RBAC92 with the addition of users (different from the subjects) and a roles hierarchy defined as a partial order. We write $\rho_{\mathsf{rbac}} = \mathsf{R}$ for the security parameter of RBAC, where R is the set of roles. A state $\sigma \in \Sigma_{\mathsf{rbac}}$ is a tuple $\sigma = (m, \mathsf{UA}, \mathsf{PA}, roles)$ where m is the set of current accesses, $\mathsf{UA} \subseteq \mathcal{S} \times \mathsf{R}$ is the relation specifying which subject can activate which roles, $\mathsf{PA} \subseteq (\mathcal{O} \times \mathcal{A}) \times \mathsf{R}$ is the relation associating permissions (i.e. pairs $(o, a) \in \mathcal{O} \times \mathcal{A}$) to roles, and $roles : \mathcal{S} \rightarrow \wp(\mathsf{R})$ specifies the set of roles that have been activated by a subject. Hence, a subject may endorse many roles, as defined by UA, but does not have to activate all of them at the same time. The RBAC policy is specified by the predicate Ω_{rbac} as follows. Given a state $\sigma = (m, \mathsf{UA}, \mathsf{PA}, roles)$, $\Omega_{\mathsf{rbac}}(\sigma)$ holds iff the two following properties are satisfied.

$$\begin{array}{l} \forall s \in \mathcal{S} \quad \{(s, r) \mid r \in roles(s)\} \subseteq \mathsf{UA} \\ \forall s \in \mathcal{S}\ \forall o \in \mathcal{O}\ \forall a \in \mathcal{A} \quad (s, o, a) \in m \Rightarrow \exists r \in \mathsf{R}\ (r \in roles(s) \wedge ((o, a), r) \in \mathsf{PA}) \end{array}$$

We write $\mathbb{P}_{\mathsf{rbac}}[\rho_{\mathsf{rbac}}] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma_{\mathsf{rbac}}, \Omega_{\mathsf{rbac}})$ for the RBAC policy.

2.2 Access Control Model

As we said previously, a language of requests provides to the subjects of a system a way to access to objects. We write $\mathcal{R}$ for the set of requests. Most access control models consider at least the set $\mathcal{R}^{acc} = \{\langle +, s, o, a\rangle, \langle -, s, o, a\rangle\}$ allowing to express that the subject s asks to get (+) or to release (-) an access over the object o according to the access mode a. Depending of the access control model, there can also exist some "administrative" requests allowing to modify security functions of a state. We introduce here the requests allowing to change the active roles of a subject: $\mathcal{R}^{adm} = \{\langle +, s, r\rangle, \langle -, s, r\rangle\}$. The set of requests considered is then $\mathcal{R}^{rbac} = \mathcal{R}^{acc} \cup \mathcal{R}^{adm}$. We make here a clear distinction between accesses and requests. An access is the internal representation of actions currently done in the system and is authorized or not according to the security policy. A request is an action that a subject has to submit and is granted or not by an implementation. However, requests are usually strongly related to accesses, and to make explicit this relation, we introduce a notion of "weak" semantics of requests as a relation $[\![\mathcal{R}]\!]_{\Sigma} \subseteq \mathcal{R} \times \Sigma$. Given a request R and a state σ, the statement $(R, \sigma) \in [\![\mathcal{R}]\!]_{\Sigma}$ characterizes the properties that a state σ must satisfy when it is obtained by applying (in a successful way) the request R over another state. For $\mathcal{R}^{rbac}$, we can define $[\![\mathcal{R}^{rbac}]\!]_{\Sigma}$ as follows:

$$
\begin{aligned}
&(\langle +, s, o, a\rangle, \sigma) \in [\![\mathcal{R}^{rbac}]\!]_{\Sigma} \Leftrightarrow (s, o, a) \in \Lambda(\sigma)\\
&(\langle -, s, o, a\rangle, \sigma) \in [\![\mathcal{R}^{rbac}]\!]_{\Sigma} \Leftrightarrow (s, o, a) \notin \Lambda(\sigma)\\
&(\langle +, s, r\rangle, (m, \mathsf{UA}, \mathsf{PA}, roles)) \in [\![\mathcal{R}^{rbac}]\!]_{\Sigma} \Leftrightarrow r \in roles(s)\\
&(\langle -, s, r\rangle, (m, \mathsf{UA}, \mathsf{PA}, roles)) \in [\![\mathcal{R}^{rbac}]\!]_{\Sigma} \Leftrightarrow r \notin roles(s)
\end{aligned}
$$

where $\Lambda(\sigma)$ denotes the set of all current accesses in σ. Note that such an approach to express a part of the semantics of requests only specifies the properties that a state must satisfy but does not describe how such a state has been changed. We introduce in [14,21] a semantical characterisation of such modifications. Due to space limitation, we omit here this technical part which is not essential at this level of specification.

Given a security parameter ρ, an access control model $\mathbb{M}[\rho]$ is defined by a tuple $\mathbb{M}[\rho] = (\mathbb{P}[\rho], [\![\mathcal{R}]\!]_{\Sigma})$ where $\mathbb{P}[\rho] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma, \Omega)$ is an access control policy, $\mathcal{R}$ is a set of requests, and $[\![\mathcal{R}]\!]_{\Sigma} \subseteq \mathcal{R} \times \Sigma$ is a relation specifying the semantics of requests. For example, we write $\mathbb{M}_{\mathsf{rbac}}[\rho_{\mathsf{rbac}}] = (\mathbb{P}_{\mathsf{rbac}}[\rho_{\mathsf{rbac}}], [\![\mathcal{R}^{rbac}]\!]_{\Sigma_{\mathsf{rbac}}})$ for the RBAC model.

Implementing a model $\mathbb{M}[\rho]$ consists in defining both a set Σ_I of *initial states* and a *transition function* $\tau : \mathcal{R} \times \Sigma \rightarrow \mathcal{D} \times \Sigma$ (where $\mathcal{D} = \{\mathsf{yes}, \mathsf{no}\}$ are the answers) which allows moving from a state to another state of the system according to a request in $\mathcal{R}$. We write (τ, Σ_I) for such an implementation and $\Gamma_\tau(E)$ for the set of reachable states by τ from states occurring in E. For example, given the set of initial states $\Sigma_I^{\mathsf{rbac}} = \{\sigma \in \Sigma_{\mathsf{rbac}} \mid \Lambda(\sigma) = \emptyset\}$, we introduce the implementation $(\tau_{\mathsf{rbac}}, \Sigma_I^{\mathsf{rbac}})$ of $\mathbb{M}_{\mathsf{rbac}}[\rho_{\mathsf{rbac}}]$ where τ_{rbac} is defined in table 1 and where we use the following denotations:

$$
(roles \oplus (s', r))(s) = \begin{cases} roles(s) \cup \{r\} & \text{if } s = s' \\ roles(s) & \text{otherwise} \end{cases}
$$

$$
(roles \ominus (s', r))(s) = \begin{cases} roles(s) \setminus \{r\} & \text{if } s = s' \\ roles(s) & \text{otherwise} \end{cases}
$$

Due to the huge number of states, we do not draw here the corresponding automaton. In [21,10], this implementation is proved to be correct according to both the policy and the semantics of requests. More formally, we prove that each state reachable from an initial state is secure (i.e. $\Gamma_\tau(\Sigma_I) \subseteq \{\sigma \in \Sigma \mid \Omega(\sigma)\}$) and that for all $\sigma_1, \sigma_2 \in \Sigma$, and $R \in \mathcal{R}$, if $\tau(R, \sigma_1) = (\mathsf{yes}, \sigma_2)$, then $(R, \sigma_2) \in [\![\mathcal{R}]\!]_{\Sigma}$.

This definition of the RBAC model within our formal framework ensures that when a request is authorized, then the security policy is not violated. An implementation of this model is defined in Focal [10], which is a IDE combining a functional language, a specification language and a theorem prover. We now want to implement it following a component-based approach, and we use the rCOS methodology, which we introduce in the next section.

Table 1. Implementation of the RBAC Model

$$\tau_{\mathsf{rbac}}(R, (m, \mathsf{UA}, \mathsf{PA}, roles))$$

$$= \begin{cases} (\mathsf{yes}, (m \cup \{(s,o,a)\}, \mathsf{UA}, \mathsf{PA}, roles)) \\ \text{if } R = \langle +, s, o, a\rangle \\ \wedge\ \exists r \in \mathsf{R} \quad r \in roles(s) \wedge ((o,a), r) \in \mathsf{PA} \\ \\ (\mathsf{yes}, (m \setminus \{(s,o,a)\}, \mathsf{UA}, \mathsf{PA}, roles)) \\ \text{if } R = \langle -, s, o, a\rangle \\ \\ (\mathsf{yes}, (m, \mathsf{UA}, \mathsf{PA}, roles \oplus (s,r))) \\ \text{if } R = \langle +, s, r\rangle \\ \wedge\ (s,r) \in \mathsf{UA} \\ \\ (\mathsf{yes}, (m, \mathsf{UA}, \mathsf{PA}, roles \ominus (s,r))) \\ \text{if } R = \langle -, s, r\rangle \\ \\ (\mathsf{no}, (m, \mathsf{UA}, \mathsf{PA}, roles)) \quad \text{otherwise} \end{cases}$$

3 Models and Their Refinement and Composition

For a formal method and its tool support to be practically effective, it will have to be integrated with a *development process* and CASE tools, such as MasterCraft [26,18]. For these purposes, the rCOS semantic theory defines the important concepts and artifacts in the domain of object-oriented and component-based software engineering, like *classes, objects, components, interfaces, contracts*, *composition* (*connectors*), *coordination* and *glue*. It provides the behavioral semantics of these concepts with high level rules for refinement and verification.

Interfaces, Contracts and Components. Component-based software engineering creates new software by combining prefabricated components with programs that provide both glue between the components, and new functionality [5]. Furthermore, there seems to be no disagreement on the following interrelated properties that *components* enjoy.

1. *Black-box composability*, *substitutability* and *reusability*: "a component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third party composition" [25].
2. *Independent development*: components can be designed, implemented, verified, validated and deployed independently.
3. *Interoperability*: components can be implemented in different programming languages and paradigms; but they can be composed, be glued together and cooperate with each another. These features require that a component has a black-box specification of what it provides to and what it requires from its environment [25].

Components and Processes. We distinguish *service components* from *process components* [12,3]. A service component, simply called a component, provides computational services to the environments through their *provided interfaces.* However, the implementation of a provided service may also require services from other components. Thus, a component can have *required interfaces*, and a component with required interfaces is called an *open components* and one without required interfaces is called a *closed component.* A distinct feature of the rCOS definition of a component is that contracts are associated to the provided interfaces and the required interfaces separately. This separation makes the specification of a component a *truly black-box specification*, even without the need to know the information about the temporal dependency between a provided service and a required service.

A process component, simply called a process, does not provide services to other components. Instead it *coordinates* and *glues* components so that the service components become suitable for a specific application. Therefore, a process only has *required interfaces* and it *actively* invokes services of other components. A component on the other hand, though it may contain coordinating processes inside it, is passive and only interacts with the outside when a provided service is requested. We will see that compositions among components are different from their compositions with processes and compositions of processes [3]. We also proved in [3] the composition of a component and a process is a component.

In rCOS, a process is used to model programs that coordinate and schedule services of components, programs that are used to *glue* components together to make new components, and to model application tasks that are realized by requesting services from components.

Contracts of Interfaces. An *interface* provides the *syntactic type information* for an interaction point of a component. It consists of two parts: the *fields declaration section*, that introduces a set of variables with their types, and the *method declaration section*, that defines a set of method signatures. Each signature is of the form $m(T_1\ in; T_2\ out)$, where T_1 and T_2 are type names, *in* stands for an input parameter, and *out* stands for an output parameter.

Current practical component technologies provide syntactical aspects of interfaces only and leave the semantics to informal conventions and comments. This is obviously not enough for rigorous verification and validation. For this, we define the notion of *contracts* of interfaces.

The *contracts* of the interfaces of a component describe what is needed for the component to be *used* in building and maintaining software systems. The description of an interface must contain information about the viewpoints among, for example *functionality*, *behavior*, *protocols*, *safety*, *reliability*, *real-time*, *power*, *bandwidth*, *memory consumption* and *communication mechanisms*, that are needed for composing the component in the given architecture for the application of the system. However, this description can be incremental in the sense that newly required properties or view points can be added when needed according to the application [11].

In the current version of rCOS, a *contract of an interface* specifies the semantics of the interface:

– The *initial condition* defines the allowable starting states.
– The *functionality specification* of each method *op* is a *reactive design* of the form $g\&p \vdash R$. In Hoare and He's UTP [13], g is called the *guard* for a *synchronization* with the environment, p is called the *precondition* and R the *postcondition* of the design. An invocation to *op* when the guard is false will be blocked. When the guard is true, the execution will take place and terminate in a state satisfying the postcondition R if the precondition p holds, otherwise the execution diverges.
– The *interaction protocol*, specifies traces of method invocations, for the environment to follow when interacting with component via the interface.

The *domain of the reactive designs* forms a complete lattice with the predicate implication as the partial order, and it is closed under the convention programming compositions of sequential composition, condition choice, non-deterministic choice and the fixed point of iteration. These compositions are also monotonic.

A contract has a *failure-divergence* semantics with that the refinement relation between contracts is defined in the same way as CSP refinement under this semantics [22]. A complete proof technique using upwards and downwards simulation is established [3].

We can divide the fields (that are the state variables) of an interface into *data variables* and *control flow variables*, the reactive designs can be decomposed into *design* of the synchronisation control the design of *data functionality*. The designs of the flow of control are *reactive designs* about the change of control states, and the designs of the data functionality are simply pre and post conditions.

4 Access Control Component

Defining a reference monitor component from the previous formalisation requires a slight adaptation. Indeed, most of the concepts introduced in section 2 are defined in a formal way, using a set-based denotation and a functional approach. Since rCOS follows an object-oriented approach, we need to adapt these concepts. Roughly speaking, we first introduce a new class for every set, as described in figure 1. Rather than defining functions to associate roles to a subject, we use the object-oriented approach by defining a UML association between the class `Subject` and the class `Role`, with the target `roles: Role[*] {unique}`. Note that we define the relation UA as an attribute of the class `Subject` rather than as another association, to clarify the presentation. We define the permissions by introducing the method `pa(r:Role, m:Mode ; ret:`*boolean*`)`, where `r` and `m` are input parameters and `ret` is the output parameter, in the class `Object`: `o.pa(r,m;ret)` will set `ret` to `true` if the role `r` can access the object `o` according to the access mode `m`.

Moreover, a class is defined for every kind of request and as a direct consequence, the reference monitor contains four different methods, that we will refer

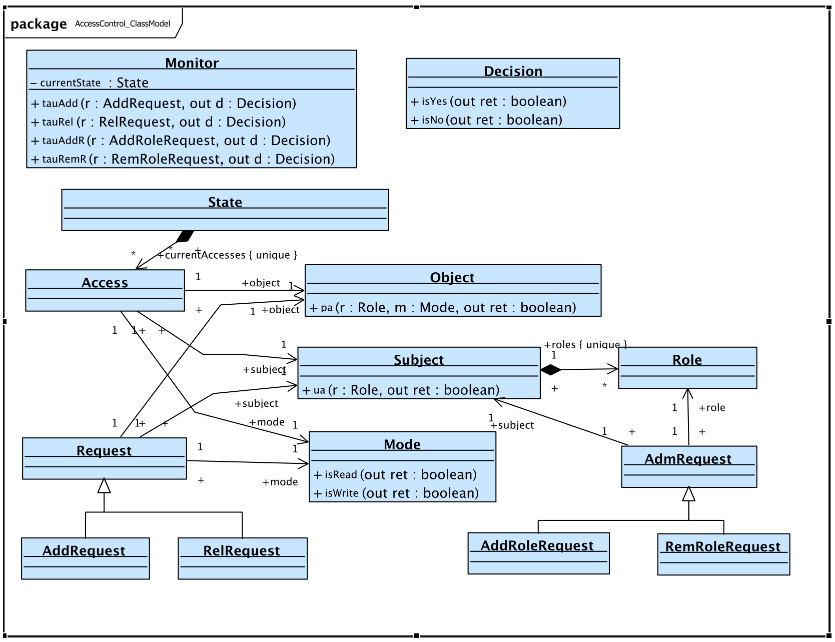
package AccessControl_ClassModel
Monitor
– currentState : State
+ tauAdd (r : AddRequest, out d : Decision)
+ tauRel (r : RelRequest, out d : Decision)
+ tauAddR (r : AddRoleRequest, out d : Decision)
+ tauRemR (r : RemRoleRequest, out d : Decision)
Decision
+ isYes (out ret : boolean)
+ isNo (out ret : boolean)
State
+currentAccesses { unique }
Access
Object
+ pa (r : Role, m : Mode, out ret : boolean)
Subject
+ ua (r : Role, out ret : boolean)
+roles { unique }
Role
+object
+subject
+mode
+role
Mode
+ isRead (out ret : boolean)
+ isWrite (out ret : boolean)
Request
AdmRequest
AddRequest
RelRequest
AddRoleRequest
RemRoleRequest

Fig. 1. Class Diagram in the rCOS Modeler

to below as the τ methods, each one of them treating a different type of request. The set of current accesses belongs to the internal state of the monitor and so we remove the reference to states in the parameters of the τ methods, which take a request `r` and return a decision `d`.

To stick more to the formal definition, we would have to also define a global method τ, which would take any type of request and would call the appropriate method according to the type. Such a method is however not necessary, so we do not include it in the interface `ValReq`.

```
component Monitor {
  provided interface ValReq {
    public tauAdd(AddRequest r ; Decision d);
    public tauRel(RelRequest r ; Decision d);
    public tauAddR(AddRoleRequest r ; Decision d);
    public tauRemR(RelRoleRequest r ; Decision d); }}
```

The controller class of this component (i.e. the class implementing the interface) is the class `Monitor`, which contains the current state, itself containing the set of current accesses. The security predicate is expressed as a class invariant, which ensures that the set of current accesses is always correct.

```
public class Monitor {
  public State currentState;
  invariant :    ∀ Subject s, ∀ Role r ∈ s.roles: s.ua(r)
               ∧ ∀ Access a ∈ currentState.currentAccesses:
                 ∃ Role r ∈ a.subject.roles: a.object.pa(r, a.mode);
```

Note that we use here the simplified notation of rCOS about output parameters: when there is only one output parameter, it can be considered as a return parameter like in usual programming languages. In the same way, to ease the reading of this paper, we use mathematical denotations for the logical connectors and the set operations instead of the rCOS keywords.

We are now in position to specify in the class `Monitor` the four τ methods defined in the interface `ValReq`. We specify them to respect the semantics of requests.

```
public tauAdd(AddRequest r ; Decision d) {
[ ⊢ d'.isYes() ⇒ ∃ Access a ∈ currentState'.currentAccesses:
                   r.subject = a.subject ∧ r.object = a.object ∧ r.mode = a.mode]
}
```

where `d'` denotes the value of the variable `d` after the evaluation of the method.

```
public tauRel(RelRequest r ; Decision d) {
[ ⊢ d'.isYes() ⇒ ∀ Access a ∈ currentState'.currentAccesses:
                   r.subject ≠ a.subject ∨ r.object ≠ a.object ∨ r.mode ≠ a.mode]
}

public tauAddR(AddRoleRequest r ; Decision d) {
[ ⊢ d'.isYes() ⇒ r.role ∈ r.subject.roles']
}

public tauRemR(RemRoleRequest r ; Decision d) {
[ ⊢ d'.isYes() ⇒ r.role ∉ r.subject.roles']
}
```

The notation $[\vdash p]$ stands for the design with the precondition `true` and the postcondition p. This specification ensures that the methods `tauAdd`, `tauRel`, `tauAddR` and `tauRemR` behave correctly according to the requests. Moreover, the separation of the definition of the security policy from the specification of these methods eases the reusability of the component. Indeed, it is possible to modify the security policy without changing this specification, if the considered requests are the same. This approach introduces a level of indirection, since these methods are not specified to respect the security policy, but rather to change the set of current accesses according to the requests, and this set is required to respect the security policy by the invariant.

Note that we present here a simple design for a reference monitor, in the sense that there is no need for defining a protocol between these methods. However, it is possible to define several global τ functions, each one of them applied in a different "security mode".

For instance, let us consider that our component relies upon an authentication mechanism (as a login/password system). If an attack is detected against this mechanism, like a brute-force attack, where an opponent tries every possible password for a given login, the system could decide to prevent any risk by denying any access asked by non-root subjects (we consider here that the root user cannot connect from the outside and so it is less prone to a brute force attack). In this case, a special mode "brute-force attack detected" could be switched on, and the method used to authorize accesses would be a more restrictive one. Such an approach would imply to define a protocol for the interface, corresponding to the several modes that could be switched on. Here again, specifying the policy as an invariant helps, since it is defined only once, and as long as the different methods `tau` respect the semantics of requests, they also respect the policy.

The definition of the different methods in the rCOS tool according to the formal definition given in table 1 and both refining the previously defined design and respecting the class invariant could be the following one.

```
public tauAdd(AddRequest r ; Decision d) {
  for (Role ro : r.subject.roles){
      if (r.object.pa(ro, r.mode))
      then {
        currentState.currentAccesses.add(r.subject, r.object, r.mode);
        return yes;
      }
  }
  return no;
}


public tauRel(RelRequest r ; Decision d) {
  currentState.currentAccesses.remove(r.subject, r.object, r.mode);
  return yes;
}


public tauAddR(AddRoleRequest r ; Decision d) {
   if (r.subject.ua(r.role)){
      r.subject.roles.add(r.subject);
      return yes;
   }
  return no;
}


public tauRemR(RemRoleRequest r ; Decision d) {
  r.subject.roles.remove(r.subject);
  return yes;
}
```

These implementations have been obtained using the rCOS refinement techniques [4,19]. They are proved to be correct by showing that the predicate corresponding to their semantics logically implies the previous specifications. Current work in the rCOS tool consists in integrating a theorem prover, in order to formally prove this implication.

Since we do not require the monitor to be complete, that is, to accept every correct request, the statement `return no` is also a valid implementation for each of the previous methods.

Integration. The reference monitor component described in the previous subsection acts as an oracle: its interface allows submitting a request which is either granted or denied. The set of current accesses stored in the internal state is only used to describe the policy. Indeed, some policies (e.g. Bell and LaPadula [16] or the Chinese Wall [1]) are defined according to the current accesses, to avoid some forbidden flows of information between objects. But in most of the cases, the interface of the reference monitor is not directly called by the user of the system. For instance, let us consider a Database Management System (DBMS), where the subjects of the monitor are the users of the database and the objects are the tables. It is also possible to consider an object as a tuple, but it can raise some problems with polyinstantiation [23]. A user executes an SQL query, which should be translated in a request $\langle +, s, o, a \rangle$, where s is the subject associated with the user (usually known in the environment from the connection), o is the concerned table and a is defined according to the query.

A DBMS can be described as the following component (we consider here only the SQL queries `INSERT, SELECT, UPDATE`).

```
component DBMS {
  provided interface SQLQuery {
    public connect (string id, string password; string con);
    public select  (string con, string query; boolean ok);
    public insert  (string con, string query; boolean ok);
    public update (string con, string query; boolean ok);
    public disconnect (string con);
  }
}
```

The method `connect` allows a user to connect to the DBMS using his login/password, and get a connection identifier, the methods `select, insert` and `update` allow to respectively execute a `SELECT`, an `INSERT` and an `UPDATE` SQL query over a connection identifier and `disconnect` closes a connection identifier.

The problem is the following one: we want to keep this interface for the user, in order to be transparent, and at the same time filter the SQL queries in order to grant only the ones respecting the security policy. We introduce a component `Proxy` as described in figure 2, which composes the `Monitor` and `DBMS` components (and hides their interfaces) and provides the same interface as `DBMS`.

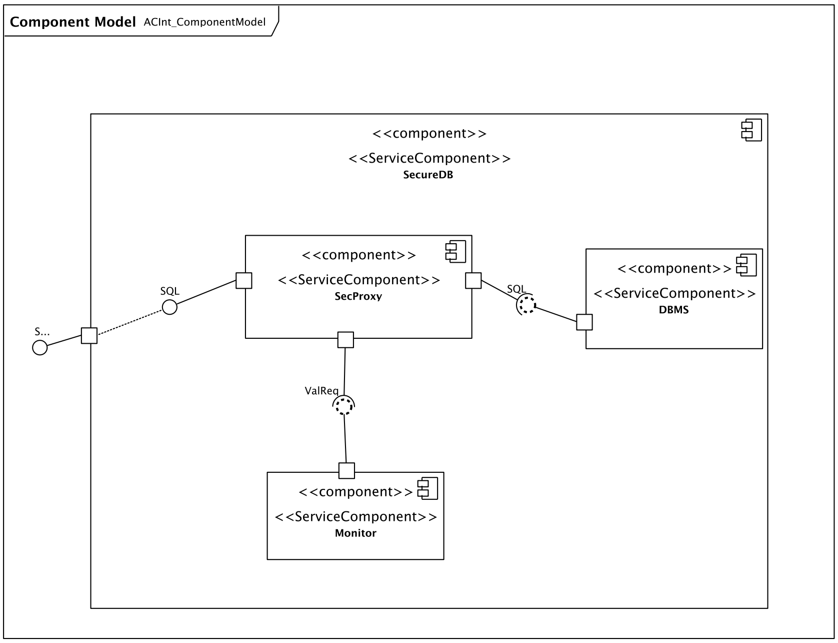

Fig. 2. Integration of a monitor in a database

```
component Proxy {
  provided interface SQLQuery { ... }
  composition : (Monitor || DBMS) \ { DBMS.SQLQuery, Monitor.ValReq }
}
```

The controller class of `Proxy` implements the interface `SQLQuery` and the pseudo-code for the method `select` is the following one.

```
public select(string con, string query; boolean ok) {
  Subject s := subject associated with the user;
  Object o := table concerned by the query;
  if (Monitor.valReq.tauAdd(⟨+, s, o, r⟩) = yes)
  then return DBMS.SQLQuery.select(con, query)
  else return false;
}
```

The method `insert` (resp. `update`) is defined in a similar way, except that the request passed to `tauAdd` is $\langle +, s, o, \mathsf{w}\rangle$ (resp. both $\langle +, s, o, \mathsf{r}\rangle$ and $\langle +, s, o, \mathsf{w}\rangle$).

With the previous definition, the accesses are never released, which is not a problem with the usual RBAC model, but which could be one with other policies such as the one of Bell and LaPadula. To address this issue, it is possible to release the accesses either at the end of the method, after the call to the DBMS or when the user disconnects from the DBMS.

5 Conclusion

By following a component-based approach to design and implement an access control reference monitor, we address some major issues as stated by the Common Criteria. Indeed, the mechanisms of composition and interfaces hiding allow to have a real black-box component, thus preventing unsafe subjects to interfere with it or to circumvent its controls. Moreover, the definition of a `Proxy` component makes its use transparent and easily integrable in a larger system, as a Database Management System. Our development relies upon a sound formal definition, which guarantees the correctness of the specification, since the rCOS tool allows integrating the formal aspect and will, in the future, use external tools, like model-checking or theorem proving, to verify and validate that the implementation meets the specification. Indeed, the proof of the correctness of the monitor is currently only done "on the paper", by induction over the reachable states. However, an objective of the rCOS Tool is to generate JML [17] specifications, and by using a tool like Krakatoa [8], which generates the proof obligations related to pre- and post-conditions and to class invariants, we could prove the correctness of the monitor with a theorem prover.

We have implemented here the RBAC policy, but thanks to the formal framework our work is based on, this approach could be used to define other policies, some of them are even already defined within this framework (e.g. Bell and LaPadula, the Chinese Wall, RBAC96, Delegation-Based, Lampson, ACL, Capabilities).

There are several ways to extend this work. For instance, nothing is said about the way to associate subjects and objects with roles in the example. Though it is not possible to define them in the most general case, it could be possible to determine them from use cases [6].

Finally, from a more practical point of view, a library of access control components could be defined. Such a library could allow software engineers with no experience in security and/or formal methods to easily use and enforce certified reference monitors.

Acknowledgements

Many thanks to Julien Blond and Mathieu Jaume for their precious help on this subject. This work was supported by the project HTTS funded by the Macao Science and Technology Development Fund and by the projects NSFC-60673114 and 863 of China 2006AA01Z165.

References

1. Brewer, D.F.C., Nash, M.J.: The Chinese wall security policy. In: Proc. IEEE Symposium on Security and Privacy, pp. 206–214 (1989)
2. Common Criteria for Information Technology Security Evaluation, `http://www.commoncriteriaportal.org/`
3. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
4. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Fitzgerald, et al. (eds.) [9]
5. de Alfaro, L., Henzinger, T.: Interface automata. In: Proc. of the 9th Annual Symposium on Foundations of Software Engineering, pp. 109–120. ACM press, New York (2001)
6. Fernandez, E.B., Hawkins, J.C.: Determining role rights from use cases. In: RBAC 1997: Proc. of the second ACM workshop on Role-based access control, pp. 121–125. ACM, New York (1997)
7. Ferraiolo, D.F., Kuhn, D.R.: Role-based access control. In: Proceedings of the 15th National Computer Security Conference (1992)
8. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: 19th International Conference on Computer Aided Verification. Springer, Berlin (2007)
9. Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.): Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, number CS-TR-1099 in Technical Report Series. Newcastle University (May 2008)
10. Habib, L.: Formalisation, comparaison et implantation d'un modèle de contrôle d'accès à base de rôles. Master's thesis, UPMC, Paris, France (2007)
11. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005)
12. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. 160, 173–195 (2006)
13. Hoare, C., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
14. Jaume, M., Morisset, C.: On specifying, implementing and comparing access control models. A Semantical Framework. Technical report, Univ. Paris 6, LIP6 (2007)

15. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
16. LaPadula, L., Bell, D.: Secure Computer Systems: A Mathematical Model. Journal of Computer Security 4, 239–263 (1996)
17. Leavens, G.T.: Jml's rich, inherited specifications for behavioral subtypes. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 2–34. Springer, Heidelberg (2006)
18. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), full version as UNU-IIST Technical Report 343 (August 2006), http://www.iist.unu.edu
19. Liu, Z., Stolz, V.: The rCOS method in a nutshell. In: Fitzgerald, et al. (eds.) [9]
20. McLean.: The algebra of security. In: Proc. IEEE Symposium on Security and Privacy, pp. 2–7. IEEE Computer Society Press, Los Alamitos (1988)
21. Morisset, C.: Sémantique des systèmes de contrôle d'accès. PhD thesis, Université Pierre et Marie Curie - Paris 6 (2007)
22. Roscoe, A.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
23. Sandhu, R., Chen, F.: The multilevel relational (mlr) data model. ACM Trans. Inf. Syst. Secur. 1(1), 93–132 (1998)
24. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer 29(2), 38–47 (1996)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1997)
26. Tata Consultancy Services. Mastercraft, http://www.tata-mastercraft.com/