A Methodology for Policy Conflict Detection Using Model Checking Techniques*

Peter H. Deussen[1], Ina Schieferdecker[1], Hiroaki Kamoda[2]

[1] Fraunhofer Research Institute for Open Communication Systems, Berlin, Germany
{deussen,schieferdecker}@fokus.fraunhofer.de
[2] NTT Data Corporation, Tokyo, Japan kamodah@nttdata.co.jp

Abstract. Security issue is an increasing concern and pressures are increasing on organizations to take a more systematic approach to incorporating security into their systems. An important component of security requirements is access control. In this paper, we introduce a variant of the *role based access control* model that assigns role to both partners of an interaction. We further develop a methodology to determine whether a set of access control policies is contradictionary, i. e, contains a conflict. Model checking techniques are employed for conflict analysis.

1 Introduction

Security issue is an increasing concern and pressures are increasing on organizations to take a more systematic approach to incorporate security into their systems. The key to this is analyzing security requirements early on, rather than treating security as an add-on, as is often the case. An important component of security requirements is access control. The goals of access control systems are to protect resources from unauthorized access and to ensure access to those resources for authorized users. There are a number of models of access control which aim to achieve this goal. The Bell-LaPadula model [14] has been particularly influential, and forms the basis of a family of multi-level security models, usually referred to as mandatory access control. Discretionary access control on the other hand has been accepted as a less rigorous way of controlling access. In this type of access control it is the owner of a resource (usually a file) who controls other users' accesses to the resource.

A promising alternative to these models is role-based access control (RBAC) [2, 16], which allows the specification of access control policy in a way that maps naturally to an organization's structure. This approach brings advantages such as easier understanding of access control policies and scalable administration. In RBAC, roles introduce a level of indirection (and abstraction) in the mapping of users and resources to privileges. Instead of mapping users and resources directly to privileges, users can assume a number of roles. In this paper, we use a variant of RBAC that allows the assignment of roles to both the user and the accessed resource of an interaction.

Security, however, is the greatest concern in access control. It was proven that the security problem (that is, verifying that a particular configuration is secure) is undecidable even for the simple access matrix [13]. Hence, access control models that enable

* This work was partially supported by Fraunhofer Research Institute for Open Communication Systems and NTT Data Cooperation

flexible expression of access control policies (like role-based ones) make the verification of whether a particular access configuration is secure (i.e. subjects do not have access to unauthorized objects) even more difficult. *Constraints* have been introduced to enable security [12], usually specified in terms of policies with negative modality. However, the coexistence of policies with different modalities inevitably leads to conflicts.

In this paper, we describe a methodology for policy conflict detection in systems with multiple distributed objects and multiple access control entities that regulate the access to services provided by those objects. The methodology comprises the following key elements:

1. The translation of a set of policies into an effective policy evaluation procedure that determines for a given user request to execute a service provided by a certain resource whether this request is granted or denied.
2. This procedure is embedded into an environment model that emulates the behavior of service requestors and providers. This environment model can be generated automatically in a canonical way from a service description (although this canonical model is not always desirable).
3. The model checker in use is SPIN [11]. The composition of the various model parts mentioned above is translated into PROMELA, the model specification language of SPIN. The resulting program is further equipped with a (apparently very simple) verification task to detect a conflict in the set of policies in question.
4. If a system under verification fails to satisfy a given property, SPIN is sometimes able to produce a so-called counterexample, that is a behavioral trace that leads from the initial system state into a state in which the property is violated. This trace—if properly displayed—provides useful information about the cause of the problem. For our purposes, we use a technique based on event and state logging to derive a meaningful representation of counterexample traces.

There are several approaches to detect conflicts in policy sets, as well as approaches that aim also on conflict resolution. In [5], a rule-based deduction method is proposed. The authors of [17] describe a problem translation into a logic program implemented in Prolog. Lupo and Sloman [15] present an approach to determine possible conflicts in RBAC by checking of role domain overlaps.

To the best knowledge of the authors, the only publication which describes the application of model checking to policy base validation is [1]. The authors apparently use also the SPIN model checker for several verification tasks of work flows in the setting of so-called *computer supported cooperative work systems*. A systematic approach to obtain validation models is not in the focus of this paper; result back interpretation is not addressed.

This paper is organized as follows. In Section 2 we introduce the access control schema that is used throughout this paper. We moreover discuss several sources of policy conflicts. Section 3 gives a very brief overview on available model checking techniques and discusses the general methodology that is employed for system validation using a model checker. We also discuss how this methodology can be applied to policy conflict detection.

In Section 4, a generic system model which builds the background of our validation activities is presented. Section 5 explains how a set of policies is translated into a

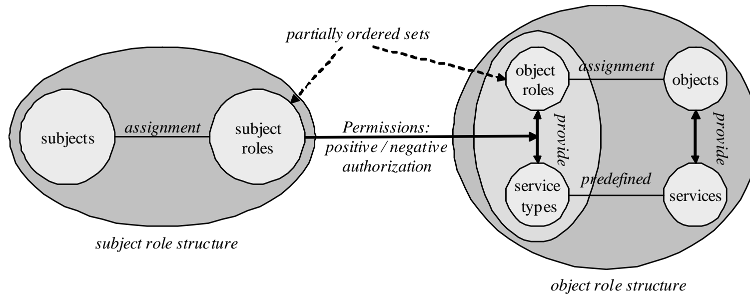

Fig. 1. Role based access control

PROMELA program. In Section 6, we address the problem how to translate information on possible conflicts from the messages provided by the model checker into a message in the terminology of the original problem statement. The final Section 7 provides a summary and an outlook on further works.

2 Role Based Access Control

Traditionally, RBAC has been designed for dealing with a relatively small number of resources (objects) to be protected. In fact, the typical application scenario is an enterprise having several thousands of employees accessing a few tens of applications. However, as we deal with network security where the number of objects can be considerably high or even not known in advance, a mean for object abstraction is needed as assigning permission to single objects become unfeasible or impossible.

Grouping of objects in domains has been introduced by M. Sloman et al. in [6] and [22] and before in [18] and integrated as part of the Ponder language [6], although without aiming at create a new complete access control model. Domains provide a flexible and pragmatic means of specifying boundaries of management responsibility and authority. A domain represents a collection of objects which have been explicitly grouped together to apply a common management policy. In the variant of role based access control that we use in this paper it is possible to assign several roles to particular individuals. Role assignment can be done both for subjects (i.e. entities that request access to certain resources) and for objects (the resources itself).

To express this in a more mathematical flavour, let us assume a finite set of *individuals* $\mathcal{I}$ and a finite sets $\mathcal{S}$ and $\mathcal{O}$ of *subject roles* and *object roles*, respectively. For convenience, we assume $\mathcal{S} \cap \mathcal{O} = \emptyset$. We moreover put $\mathcal{R} =_{\mathrm{df}} \mathcal{S} \cup \mathcal{O}$. We fix the conventions that roles are denoted by lowercase boldfaced letters $\boldsymbol{s}, \boldsymbol{o}, \boldsymbol{r}$, with possible subscripts, e. g. $\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2$. For individuals, italic letters like i, o, and s—again with possible subscripts—are used. Let $\mathbf{mbrs} : \mathcal{R} \rightarrow \mathbf{P}(\mathcal{I})$ and $\mathbf{roles} : \mathcal{I} \rightarrow \mathbf{P}(\mathcal{R})$ be functions that return the set of individuals of a particular role and the set of roles are assigned to a particular individual, respectively. Further, let $\mathcal{A}$ be a set *service identifiers*. Services are provided

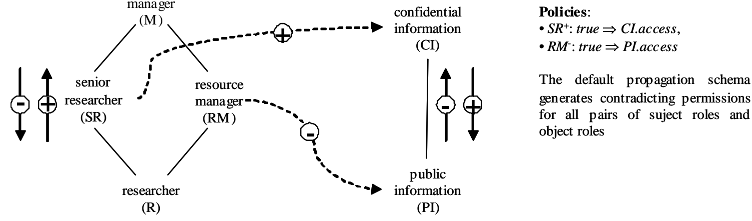

Fig. 2. A conflict caused by policy propagation

by individuals $o \in \mathcal{I}$; a concrete service associated with a service identifier $a \in \mathcal{A}$ provided by an individual $o \in \mathcal{I}$ is denoted by $o.a$.

Services execution is assumed to be guarded by some access control. This access control is defined by a set of *policy* rules of the form $\boldsymbol{s}^x : \varphi \Rightarrow \boldsymbol{o}.a$, where $x \in \{+, -\}$ is a *mode*, $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{o} \in \mathcal{O}$ are called the *subject role* and the *object role* of the policy, $a \in \mathcal{A}$ is a service identifier, and φ is a *condition*. We assume that there is a number of atomic propositions available that relates to states of the underlying controlled system (we will not explain the term "system" in a formal fashion). Conditions are then defined as Boolean expressions (constructed using the operations $\neg$, $\wedge$, and $\vee$) over those propositions.

Assume that an individual $s \in \mathcal{I}$ requests a service $a \in \mathcal{A}$ from another individual $o \in \mathcal{I}$; s is then called the *subject* and o is called the *object* of this interaction. Requests are denoted as $\mathbf{req}(s, o, a)$. Let Π be a set of policies that define the access control for the service $o.a$. A policy $\boldsymbol{s}^x : \varphi \Rightarrow \boldsymbol{o}.a \in \Pi$ is called *active* for the request $\mathbf{req}(s, o, a)$ if $s \in \mathbf{mbrs}(\boldsymbol{s})$, $o \in \mathbf{mbrs}(\boldsymbol{o})$, and φ evaluates to true. The request is called *permitted* by Π if

1. there is some active policy $\pi \in \Pi$ with mode "$+$", and
2. no policy $\pi' \in \Pi$ with mode "$-$" is active.

2.1 Policy Propagation

A propagation scheme for a set of policies Π assigns to each policy a set of roles for which this policy does also hold. Suppose two mappings $\mathbf{prgt}_{\mathcal{S}} : \Pi \rightarrow \mathbf{P}(\mathcal{S})$ and $\mathbf{prgt}_{\mathcal{O}} : \Pi \rightarrow \mathbf{P}(\mathcal{O})$ are defined such that $\boldsymbol{s} \in \mathbf{prgt}_{\mathcal{S}}(\pi)$ and $\boldsymbol{o} \in \mathbf{prgt}_{\mathcal{O}}(\pi)$ for all policies $\boldsymbol{s}^x : \varphi \Rightarrow \boldsymbol{o}.a \in \Pi$. Then Π is called *closed* against a propagation scheme if for all $\pi = \boldsymbol{s}_1^x : \varphi \Rightarrow \boldsymbol{o}_1.a \in \Pi$ we also have $\boldsymbol{s}_2^x : \varphi \Rightarrow \boldsymbol{o}_2.a \in \Pi$ for all $\boldsymbol{s}_2 \in \mathrm{prgt}_{\mathcal{S}}(\pi)$ and for all $\boldsymbol{o}_2 \in \mathrm{prgt}_{\mathcal{O}}(\pi)$.

The above definition is quite general. In practice, propagation along hierarchical structures defined in an organization is employed: Positive authorization is propagated "upward" in the subject role ordering and "downward" in the object role ordering, while negative authorization is propagated in the opposite direction (comp. Fig. 2 for an example). Hence we assume that the sets $\mathcal{S}$ and $\mathcal{R}$ are partially ordered by the relations

$\leqslant_{\mathcal{S}}$ and $\leqslant_{\mathcal{O}}$, respectively. Let $\pi = \boldsymbol{s}^x : \varphi \Rightarrow \boldsymbol{o}.a \in \Pi$ be a policy. Then the *default propagation* scheme for π in Π is defined as follows:

1. if $x = +$ then $\mathbf{prgt}_{\mathcal{S}}(\pi) =_{\text{df}} \{\boldsymbol{s}' \in \mathcal{S} : \boldsymbol{s} \leqslant_{\mathcal{S}} \boldsymbol{s}'\}$ and $\mathbf{prgt}_{\mathcal{O}}(\pi) =_{\text{df}} \{\boldsymbol{o}' \in \mathcal{O} : \boldsymbol{o}' \leqslant_{\mathcal{O}} \boldsymbol{o}\}$
2. if $x = -$ then $\mathbf{prgt}_{\mathcal{S}}(\pi) =_{\text{df}} \{\boldsymbol{s}' \in \mathcal{S} : \boldsymbol{s}' \leqslant_{\mathcal{S}} \boldsymbol{s}\}$ and $\mathbf{prgt}_{\mathcal{O}}(\pi) =_{\text{df}} \{\boldsymbol{o}' \in \mathcal{O} : \boldsymbol{o} \leqslant_{\mathcal{O}} \boldsymbol{o}'\}$

2.2 Conflicts

By a *conflict* we mean a contradicting assignment of permissions to some individual; formally: Policies $\boldsymbol{s}_1^x : \varphi_1 \Rightarrow \boldsymbol{o}_1.a \in \Pi$ and $\boldsymbol{s}_2^y : \varphi_2 \Rightarrow \boldsymbol{o}_2.a \in \Pi$ are *conflicting* in Π for some request $\mathbf{req}(s, o, a)$, if there is some system state such that both policies are active for this request, and moreover, $x \neq y$. Note that this implies $s \in \mathbf{mbrs}(\boldsymbol{s}_1) \cap \mathbf{mbrs}(\boldsymbol{s}_2)$ and $o \in \mathbf{mbrs}(\boldsymbol{o}_1) \cap \mathbf{mbrs}(\boldsymbol{o}_2)$

What are the possible sources of conflicts?

Inconsistent Direct Assignment of Access Permissions The most obvious source of conflicts is an error in the assignment of access permissions for some service. Conflicts of this type can be easily detected by a simple search. By using a lexicographical ordering of the policy base this search can be performed very efficiently. Please note that this simple algorithm is too "pessimistic" because even if we have opposite policies assigned to the same combination of subject and object roles and service, it is still possible that the conditions that control the activation of these policies never get true simultaneously.

Policy Propagation Things get more complicated if automated policy generation features like permission propagation are employed. Fig. 2 gives an example of a conflict caused by policy propagation. In the example, access permissions on "confidential information" are assigned to "senior researchers", while "research managers"—perhaps accidentally—are not allowed to access "public information". The default propagation schema applied to the object role structure now causes a conflict. But since the propagation scheme is also applied to the subject role structure, all roles are now "infected" be the conflict between these two policies.

Multiple Assignment of Roles In RBAC, multiple assignment of roles is permitted, i. e. it is possible for an individual to assume more than one role. Hence, inconsistent assignment of roles is another source of conflicts.

For an example (comp. Fig. 3), consider two service providers (SPs) A and B which offer service objects R_A and R_B to their customers, respectively (R_A and R_B are used as object roles in this example), with services *login* and *use*. Now assume that A and B decides to collaborate in order to provide a joint service object R_J (Fig. 3). While defining access control policies for this situation, a difference in the security concepts of both SPs remains unnoticed: SP A assumes that its customers (belonging to role C_A) are allowed to use R_J once they are logged in either at R_J or R_A, while SP B assumes that a separate login at R_J is necessary for its customers (belonging to role C_B). Now the conflict occurs if there is a customer $c \in \mathbf{mbrs}(C_A) \cap \mathbf{mbrs}(C_B)$ that logs in at

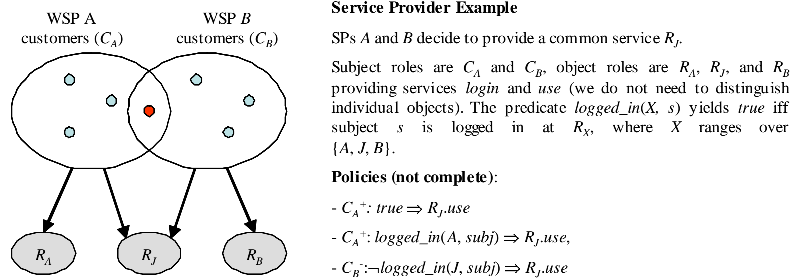

Fig. 3. A conflict caused by multiple role assingments

.

R_A. From the point of view of A, $R_J.use$ is now permitted, while it is denied from the point of view of B.

3 A Methodology for Policy Conflict Detection

In this section, we are going to discuss the key elements of a model checking methodology and give an overview on how this methodology can be applied to the problem of policy conflict detection.

3.1 Model Checking

By a *model checker* we refer to a computer program that checks for a given system specification or system model (e. g. a Petri Net, a SDL diagram, or some text in a programming language like notation) and a denotation of a system property whether this specification satisfies the property, or—stated from a more logical point of view—whether the specification is a model of the property.[3] The term “model checking” refers moreover to a certain algorithm that can be summarized as “exhaustive system simulation”, i. e. the explicit enumeration of all possible execution alternatives of the system model while simultaneously or afterwards checking for violations of the property under validation. Since the state space of a system model may be exponential (and even super-exponential) in its size, the model checking approach suffers from what is called the *state explosion problem*. There are a number of approaches to deal with this problem; we mention just a few:

Symbolic State Space Representation. A binary decision diagram (BDD) [3] is a data structure that stores bit vectors of a fixed length in a highly compressed way. The applicability to model checking was noted by McMillan [4]: a model checker for a temporal logic called *computation tree logic* (CTL) can be easily defined by using efficient BDD

[3] In the context of model checking, properties are usually given in some temporal logics. See [8] for a standard reference.

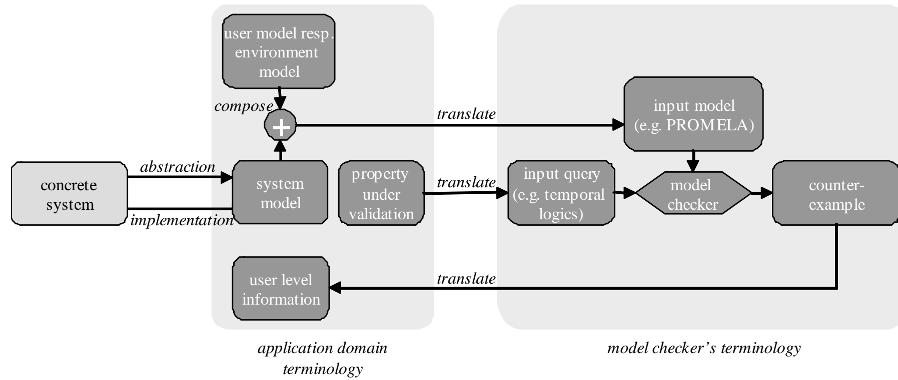

Fig. 4. Using model checking technology for system validation

operations. A prominent implementation of this approach is the model checker SMV [19].

Partial Order Techniques. In fact, there are two different (although related) approaches that deserve both the name "partial order technique". *Implicit partial order techniques* base on the idea that it is not necessary to enumerate all interleavings of concurrent system activities in order to decide a given analysis property. First model checking algorithms that base on this idea were introduced by Valmari [21] and generalized by Godefroid [10]. The model checker SPIN [11] used in this paper bases on this approach. In opposite to implicit partial order techniques, *explicit partial order techniques* use a representation of the behavior of a concurrent system that bases on partial orders. See e. g. [9, 20, 7] for applications to model checking.

3.2 Model Checking Process

How can the model checking approach applied to real system verification tasks? What is required here is not only an efficient model checking algorithm that produces results in a reasonable amount of time, but also a suitable validation process. In the following, we discuss some of the methodological key elements and how they can be applied to conflict detection in access control policy bases. For that what follows, please refer to Fig. 4.

System Model. Model checking is performed not on real systems but on system models, i. e. on abstractions. If we have a complete system specification that covers all behavioral aspects of its implementation, than we can use this specification directly as starting point of the validation process. In practice however, specifications are often partial, informal, or otherwise not suitable. It is however sometimes possible to build an abstraction from the real, implemented system (see [11] for a detailed discussion).

Environment Model and Composition. Model checking is usually applied to the verification of reactive systems., i. e. systems that maintain an ongoing interaction with their environments. Thus having just a model of the system in question is generally not enough, we also need to have a model of its environment in order to generate stimuli and to receive responses. A very simple canonical way to define an environment model is to use a number of process models that repeats to produce concurrently all possible requests and to accept any response from the system model. This however is not what is appropriate in all cases. Sometimes, the system model is partial in the sense that it makes assumptions on the behavior of the environment (although the system implementation may check for situations in which these assumptions are violated and react accordingly). Another disadvantage is that the usage of a canonical environment is likely to blow up the number of system states.

A way out is to use environment models that expresses certain expectations on the order in which the interactions with the model under validation takes place. Obviously, this approach produces now only partial results, i. e. if the production environment of the system implementation does not follow these assumptions, the implementation may behave incorrect even if it is a correct implementation of its specification. Note that those more tailored environment models can be obtained by adding further dependencies (i. e. communication) between canonical environment processes that handles particular requests.

Problem Translation. Usually, a problem statement (comprising of a system model together with a verification environment and an analysis question) is expressed in terms related to a particular domain. Thus a translation of the original system model into the language of the model checker is needed. The same holds for the specification of the property to be validated: As already mentioned, most model checkers use some type of temporal logics as property specification language. Hence, another translation process is necessary for property specifications.

Result Back Translation. Since during the translation process from the problem domain into the model checker's terminology information on the original system model is lost, the only way to enable a comprehensive result interpretation in terms of the application domain is to translate not only the system model but also enrich this translation with enough information to relate events and state components listed in the counterexample with actions of the original system actions and values.

Application to Conflict Detection. To work out the details of the application of the model checking methodology to the problem of policy conflict detection, we carried out a number of experiments using the following tools:

1. To specify policy bases, we defined a small text based description format that covers all relevant aspects, in particular role structures (including partial orders on roles), and policy definitions. Conditions has been directly defined as PROMELA expressions referring to variable values of the environment model (like the predicate "logged_in()" from example 3).

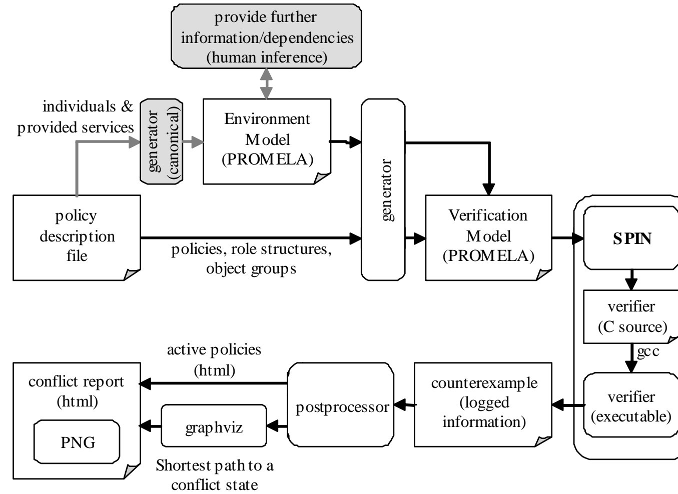

Fig. 5. Verification tool chain

2. Client and service objects with all provided services are also specifiable in this description format. For each possible request, i. e. an element of the set

$$\{\mathbf{req}(s, o, a) \mid s, o \in \mathcal{I} \ \&\ a \in \mathcal{A} \ \&\ o \text{ provides service } a\},$$

an independent PROMELA process is generated. This process just sends the associated request and waits for an answer according to the protocol explained in Section 4 below. These canonical environment processes are further enhanced by variables reflecting their current local states. This information is used in policy predicates (and could be used for further verification tasks). If a certain behavior of the environment is assumed, the sending of requests can be sequentialized by additional synchronization messages. Finally, logging directives for result back interpretation (see below) are added. Currently, these modification steps of the canonical environment model have to be done manually.
3. To allow for result back interpretation, all generated pieces of code are equipped with extensive logging information. If the model checker reports the violation of a verification task by a counterexample, then this trace can be replayed. During this replay, the logging directives are executed. Thus it is relatively easy to interpret a trace in terms of the original problem statement.

Fig. 5 shows the set of translation and validation tools that are used in our experiments. The validation process starts with a policy description file. A generator program (white box) is used to obtain a verification model file from this description as explained in Section 5. This generator uses another input file containing environment model processes which is simply appended to the output PROMELA file. Additionally, the generator can be used to generate canonical environment model processes (gray box).

Then the model checker SPIN is applied. Spin generates a verifier program (in C), which performes the actual model checking process. If a conflict is detected, a counterexample is generated, i.e. a guided execution sequence of the original input model. If these sequence is executed again (playback), a trace of the behavior that lead to the conflict is obtained. This trace is further processed by a postprocessor program that outputs a graph description processible by the GraphVis package which is used to generate a graphical presentation of the conflicting behavior in PNG format. Additionally, a HTML page containing the set of active policies in the conflicting state and the generated PNG picture is produced. A simple UNIX shell script is used to pipeline these tools.

4 System and Environment Model

In this section, we describe a generic model of a number of interacting distributed objects. Interactions are regulated by special access control components.

Recall that $\mathfrak{I}$ denotes a set of *individuals*. These individuals—human or automatic—are realized by concurrently running processes that communicate with other individuals by sending requests for the execution of certain services and receive the result of those computation by a replay message.[4] In this setting, each individual may appear both as service requestor and as a service provider. Concerning a specific interaction, we call in accordance to the terminology introduced in Section 2 the service requestor the *subject* and the service provider the *object* of this interaction.

Access authorization is performed by *access control components*. We assume that resources are grouped into pair-wise disjoint sets of objects that are guarded by the same access control component. Let AC_G such an component for a group $G \subseteq \mathfrak{I}$. Instead sending requests $\mathbf{req}(s, o, a)$ for some $o \in G$ directly to the individual o, the request is now sent to the access control component AC_G. AC_G maintains internally a set of access control policies Π_G. If $\mathbf{req}(s, o, a)$ is permitted in Π_G, AC_G sends a replay of the form $\mathbf{rep}(permit)$ back to s, and forwards the request to the service provider o which executes the requested action $o.a$ and sends the result of this computation back to s. If otherwise $\mathbf{req}(s, o, a)$ is denied in Π_G, then a replay $\mathbf{rep}(deny)$ is responded to s. The request is not forwarded. This protocol is depicted in Fig. 6.

The communication model (synchronous, asynchronous, asynchronous with faulty communication medium, etc.) that is used to realize the interactions described above clearly depends on the environment in which these interaction takes place (in the Internet, in presence of a reliable middleware, etc.), and on the level of abstraction that is chosen for the system model. In this paper, we assume that interactions are realized by asynchronous communication using an input queue model (compare Fig. 7.(a)): Each individual $i \in \mathfrak{I}$ maintains several input queues of fixed capacity (according to the finiteness assumption of the model checking approach) in which incoming messages are stored. If the capacity of a message queue is reached, then another individual $j \in \mathfrak{I}$ that uses this queue to communicate with i is blocked until i consumes another message from that queue. The special case of a queue capacity equal to zero results in a synchronous communication.

[4] We do not assume that each individual is implemented by a sequential process. In fact, individuals may perform several tasks concurrently. An example is the set of canonical environment processes for the requests that are sent by the same subject.

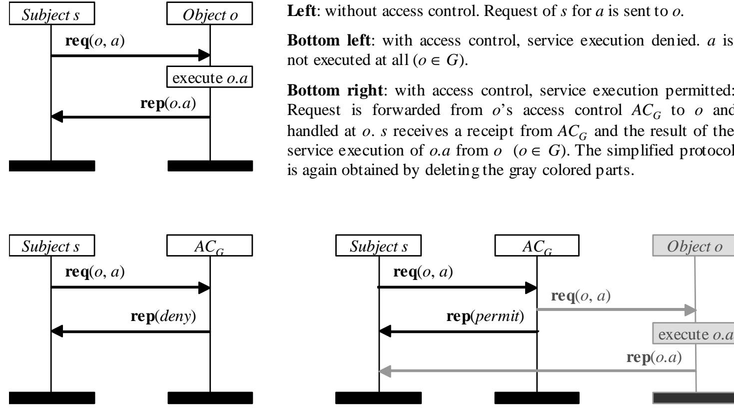

Fig. 6. Interaction protocol to request a service

This communication model can directly be mapped on the channel communication model supported by PROMELA.

Note that if all service providing individuals are *passive* in the sense that they react only on requests of other individuals without initiating actively any interaction, and moreover, they are *stateless*, i. e. their behavior do not depend on (local or global) variables or on interactions with other processes, then a simplified interaction model can be applied. This model is obtained by deleting the gray parts in the Figures 6 and 7.

5 Problem Translation into Promela

We now briefly discuss how to translate a system structured as outlined in Section 4 into POMELA. Let $\mathit{\Pi}_G$ be a set of policies for some group G of objects and let $\pi = \boldsymbol{s}^x : \varphi \Rightarrow \boldsymbol{o}.a \in \mathit{\Pi}_G$ be a policy. Recall that the mapping $\mathbf{mbrs}$ associates a set of individuals with each role $\boldsymbol{r} \in \mathcal{R}$. Further recall the definition of a propagation scheme as a pair of mappings $\mathbf{prgt}_{\mathcal{S}}$ and $\mathbf{prgt}_{\mathcal{O}}$.

Fig. 8 shows several code fragments that are the result of the translation of a set of policies and associated environment models into a PROMELA program. Unimportant parts, e. g. an initial process that initiates all other processes, are omitted; furthermore, only models for particular requests are not shown. We moreover take the freedom to leave some parts of the program implicit (in particular arrays of channels and the associated access mechanisms). Some of the code fragments are named (EXP, EVAL) for further reference; the translation procedure that has been implemented replaces these "macros" be the defining pieces of code. The meaning of the "macros" LOG and LOG_IF will be explained in Section 6.

Our program makes extensive use of the PROMELA feature to send channel identifier as parts of messages: for instance, if a process P_1 declares a local channel identifier C and sends it to another process P_2, then C can be used by P_2 to send messages back to

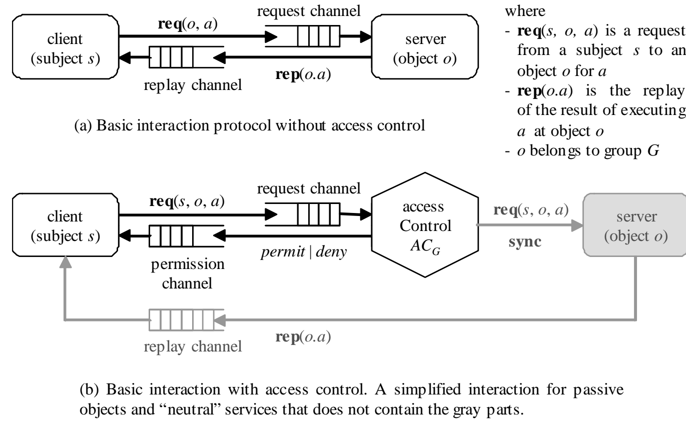

Fig. 7. Basic interaction between subjects and objects

P_1 without knowing the identity of P_1. For the sake of convenience, we assume that all message queues (if not synchronous, i.e. of capacity 0) are of the same length (say n). We moreover assume that the execution of any service yields a datum of type *result*.

Fig. 8.(a) shows a canonical subject model for a possible request. The process initiates the request for a service execution by some object o and receives the response from the access control associated with the group G to which o belongs. If this response is positive, then the process gets ready to recieve the result of the computation of the requested service from the object o.

In Fig. 8.(b), the code to determine the activation of a policy π is shown. It is checked whether the subject and object of a request belong to the respective subject and object roles of π, and whether the service identifier of the request is equal to the service identifier of π. Finally, the predicate φ is evaluated. Note that the sets $\mathbf{mbrs}(\boldsymbol{r})$, $\mathbf{prgt}_{\mathbb{S}}(\boldsymbol{s})$, and $\mathbf{prgt}_{\mathbb{O}}(\boldsymbol{r})$ are always finite. Thus the question whether an individual belongs to a given set of roles can be expanded to a Boolean expression (compare the code fragment EXP). The access control process (Fig. 8.(c)) maintains two integer variables n_+ and n_- that are used to count the number of positive and negative authorization policies that are active of a given request.

Finally, Fig. 8.(c) shows how a set of policies Π_G is embedded into an evaluation environment, namely the process *AccessControl*$_G$. The process starts with receiving a request via the globally defined request channel req_G. With the request, two channel identifier are transferred: *prep* is used to send the result of the policy evaluation process (true for *permit* and false for *deny*) back the the requesting process, while *res* is forewarded to the service providing object to transfer the result of the service computation back to the client.

The access control process continues with the evaluation of all policies using the code fragment $EVAL$ as explained before. During policy evaluation, the number of active positive and negative policies is stored in the variables n_+ and n_-, respectively.

```
(1)   chan req_G = [n] of { mtype, mtype, mtype, chan, chan }; /* One channel for each group G */

      /* "Template process" for the handling of requests req(s, o, a) */
(2)   proctype Request(mtype s, mtype o, mtype a) {
(3)     chan prep = [0] of { bit }; chan rep = [n] of { result }; result res;
(4)     do
(5)     :: LOG_s("sending request %e, %e, %e", s, o, a);
(6)        req_G!s, o, a, rep, prep;
(7)        if
(8)        :: prep?true → rep?res; LOG_s("request %e, %e, %e granted", s, o, a)
(9)        :: prep?false → LOG_s("request %e, %e, %e denied", s, o, a)
(10)       fi
(11)    od
(12)  }
```

(a) Cannonical Models of Individuals (one process for each possible request)

```
(1)   EXP(i, {i_1, i_2, ..., i_n}) ≡ (i == i_1 || i == i_2 || ··· || i == i_n)

(2)   EVAL_π(s, o) ≡
(3)     if /* Check whether policy π = s^x : φ ⇒ o.σ is active */
(4)     :: EXP(s, ⋃_{s ∈ prgt_S(π)} mbrs(s)) && EXP(o, ⋃_{o ∈ prgt_O(π)} mbrs(o)) && φ →
(5)        n_x++; LOG_AC(π)
(6)     :: else → skip
(7)     fi
```

(b) Policy Evaluation Code

```
(1)   proctype AccessControl_G() {
(2)     chan prep = [0] of { bit }; chan res = [n] of { result };
(2)     chan frwd_o = [0] of { mtype, chan }; /* One channel for each o ∈ G */
(3)     mtype s, o, a; int n_+, n_-;
(4)     do
(5)     :: req_G?s, o, a, prep, res →
(6)        atomic {
(7)          LOG_AC("request %e, %e, %e recieved", s, o, a);
(8)          n_+ = 0; n_- = 0;
(9)          EVAL_π1(s, o, a); EVAL_π1(s, o, a); ...; EVAL_πk(s, o, a);
(10)         LOG_IF_AC(n_- * n_+ != 0, "conflict found");
(11)         assert(n_- * n_+ == 0);
(12)         if
(13)         :: (n_+ > 0) → prep!true; frwd_obj!a, res; LOG_AC("replay %e, true", s);
(14)         :: else → prep!false; LOG_AC("replay %e, false", s);
(15)         fi;
(16)         LOG_AC("request processed");
(17)       }
(18)    od
(19)  }
```

(c) Access Control Process

Fig. 8. PROMELA model code

Obviously, we have a conflict if and only if $n_+ \times n_- \neq 0$ is true. We use an **assert** statement to turn this condition into a verification task for SPIN.

The request is permitted if $n_+ > 0$ & $n_- = 0$ is true (if the first part is true, the second term is assured by the preceeding **assert** statement). If this is the case, the service request is forwarded to the service providing object.

6 Interpretation of Results

Let us now discuss how to transfer information on possible conflicts back to the system administrator who is responsible to maintain a set of policy bases. It happens that PROMELA provides output directives that allows to print out a string during the replay of a counterexample. While translating the system model and composing it with the environment models, we enrich the resulting composition with suitable output statements. Thus if a counterexample is displayed, the resulting trace is expressed on domain specific level in terms of active policies, requests, and replays.

In the PROMELA code of Fig. 8, we made use of two code abbreviations, namely $\text{LOG}_v(\ldots)$ and $\text{LOG_IF}_v(c, \ldots)$. Here, v is the identification of an individual, or of an access control component. $\text{LOG}_v(\ldots)$ is expanded to code that prints out the information given in the "argument" prefixed by v, $\text{LOG_IF}_v(c, \ldots)$ does the same if the condition c evaluates to true (similar to the syntax of C language format strings, $\%e$ is used to output an identifier (of type **mtype**)). Thus while replaying a counterexample trace, SPIN prints out the interaction of individuals and access control components that has lead to a system state in which a conflict has been occurred. Since it is impossible in PROMELA that a message overtakes another one (we do not use the random receive operation ("??")), the order of messages in such a trace reflects the order in which these messages have been sent and received. Thus it is possible to reconstruct the component-local part of the interaction from the global interleaving of events reported by the LOG directives. Special treatment has to be done for messages that have been sent but have not been received at the point of time when a conflict (i. e. the violation of an **assert** statement) is detected. Fig. 9 shows an example of a conflict report produced from a counterexample trace for the conflict situation that has been discussed in Fig. 3 (the additional "logged in at A" event was produces by an additional LOG statement in the process for the *login* request).

7 Summary and Further Work

In this paper a role based access control model that provides multiple assignments of subject and object roles to individuals (service requestors and providers) has been defined. It has been shown that policy conflicts from several sources are possible in this scheme. A methodology to use model checking techniques (tailored for the model checker SPIN) has been defined. This methodology provides the definition of a canonical verification environment as well as the translation of results back into the terminology of the application domain. Although our approach works reasonable fast for small examples, a "real-world" case study has not been done yet.

Our work can be extended in several directions, we just mention a few:

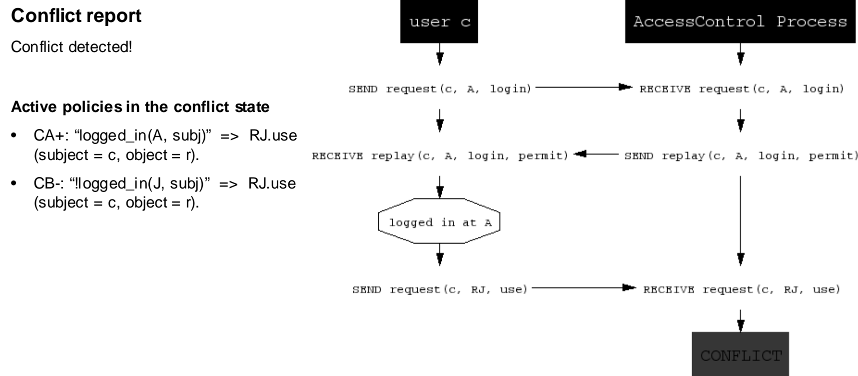

Fig. 9. Conflict report example output

1. Another type of policies not considered in this paper are *obligation policies*. An obligation policy states that under certain circumstances a subject has to perform a particular action.[5] Clearly, the subject is not allowed to defer this action, it has to perform it as soon as possible. This adds another difficulty because in a concurrent environment the notion of the next action to be performed is not clear. In particular, none of the available standard model checkers seem to be able to deal with this problem. In opposite to that, model checkers that base on an explicit partial order representation of the system behavior may provide the necessary temporal operators.
2. The example in Fig. 9 shows that the model checking approach is capable to produce very precise information on the circumstances under which a policy conflict can occur. A conflict resolution method based on this approach could use this information to resolve a conflict exactly in those system states in which it occurs without modifying the set of policies under consideration more than necessary.
3. In the current setting, the set of policies cannot be modified during the validation process. A dynamic environment however may implement computation and enforcement of new policies (the easiest example is an individual that changes the permissions of another one). PROMELA provides the necessary data structures (arrays and structures), but details have not been worked out yet.

References

1. Ahmed, T. and A. Tripathi: 2003, ‘Static Verifi cation of Security Requirements in Role Based CSCW Systems’. In: *In 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*. pp. 196–203.
2. Barkley, J. F., A. V. Cincotta, D. F. Ferraiolo, S. Gavrila, and D. R. Kuhn: 1997, ‘Role Based Access Control for the World Wide Web’. In: *Proc. 20th NIST-NCSC National Information Systems Security Conference*. pp. 331–340.

[5] A negative form of obligation policies also exists, a discussion of the precise difference to negative authorization policies is however beyond the scope of this paper.

3. Bryant, E. R.: 1992, ‘Symbolic Boolean Manipulation with Ordered Binary-Dicision Diagrams’. *ACM Computing Survey* **24**(3), 293–318.
4. Burch, J. R., E. M. Clarke, and K. L. McMillan: 1990, ‘Symbolic Model Checking: 10^{20} States and Beyond’. In: *The 5th Annual IEEE Symposium on Logic in Computer Science*. New York, pp. 428 – 439.
5. Cholvy, L. and F. Cuppens: 1997, ‘Analyzing Consistency of Security Policies’. In: *RSP: 18th IEEE Computer Society Symposium on Research in Security and Privacy*. pp. 103–112.
6. Damianou, N., N. Dulay, E. Lupu, and M. Sloman: 2001, ‘The Ponder Policy Specifi cation Language’. *Lecture Notes in Computer Science* **1995**, 18–38.
7. Deussen, P. H.: 2001, ‘Analyse verteilter Systeme mit Hilfe von Prozessautomaten’. Ph.D. thesis, Brandenburg Technical Univ. at Cottbus. avail. only in German.
8. Emerson, E. A.: 1990, ‘Temporal and Model Logic’. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*. Elsevier Science Publishers B.V., Chapt. 16, pp. 997 – 1072.
9. Esparza, J.: 1994, ‘Model checking using net unfoldings’. *Science of Computer Programming* **23**, 151–195.
10. Godefroid, P.: 1996, *Partial-Order Methods for the Verification of Concurrent Systems*, No. 1032 in Lecture Notes in Computer Science. Springer-Verlag.
11. Holzmann, G. J.: 2003, *The SPIN Model Checker — Primer and Reference Manual*. Addison-Wesley.
12. Jeager, T.: 2001, ‘Managing access control complexity using metrices’. In: *Proceedings of the sixth ACM symposium on Access control models and technologies*. pp. 131–139.
13. Lampson, B.: 1971, ‘Protection’. In: *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*. Princeton University, pp. 437–443.
14. LaPadula, L. J. and D. E. Bell: 1973, ‘Secure Computer Systems: A Mathematical Model’. Technical Report 2547, Vol. II, MITRE.
15. Lupu, E. and M. Sloman: 1999, ‘Conflicts in Policy-based Distributed Systems Management’. *IEEE Transactions on Software Engineering — Special Issue on Inconsistency Management* **25**(6), 852–869.
16. Moffett, J. D.: 1998, ‘Control Principles and Role Hierarchies’. In: *ACM Workshop on Role-Based Access Control*. pp. 63–69.
17. Ribeiro, C., A. Zuquete, P. Ferreira, and P. Guedes: 2000, ‘Security Policy Consistency’. In: *Workshop on Rule-Based Constraint Reasoning and Programming*.
18. Sloman, M. and K. Twidle: 1994, ‘Domains: A Framework for Structuring Management Policy’. In: *Network and Distributed Systems Management.* Addison Wesley, pp. 433–453.
19. The SMV System. Further information and download avail. at http://www-2.cs.cmu.edu/ modelcheck/smv.html.
20. Thiagarajan, P. S. and J. G. Henriksen: 1998, ‘Distributed Versions of Linear Time Temporal Logic: A Trace Perspective’. In: W. Reisig and G. Rozenberg (eds.): *Lectures on Petri Nets I: Basic Modells*, No. 1491 in Lecture Notes in Computer Science. Springer-Verlag, pp. 643 – 679. Lecture Notes of the 3rd Advanced Course on Petri Nets, Dagstuhl (1996).
21. Valmari, A.: 1992, ‘Stubborn Attack on State Explosion’. *Formal Methods in System Design* **1**, 297–322.
22. Yialelis, N. and M. Sloman: 1996, ‘A Security Framework Supporting Domain Based Access Control in Distributed Systems’. In: *ISOC Symposium on Network and Distributed Systems Security (SNDSS96)*. San Diego, California, pp. 26–39.