A Compositional Framework for Access Control Policies Enforcement

François Siewe
fsiewe@dmu.ac.uk

Antonio Cau
acau@dmu.ac.uk

Hussein Zedan
hzedan@dmu.ac.uk

Software Technology Research Laboratory
De Montfort University
The Gateway, Leicester
LE1 9BH, UK

ABSTRACT

Despite considerable number of work on authorization models, enforcing multiple policies is still a challenge in order to achieve the level of security required in many real-world systems. Moreover current approaches address security settings independently, and their incorporation into systems development lifecycle is not well understood. This paper presents a formal model for the specification of access control policies. The approach can handle the enforcement of multiple policies through policies composition. Temporal dependencies among authorizations can be formulated. Interval Temporal Logic (ITL) is our underlying formal framework and policies are modeled as safety properties expressing how authorizations are granted over time. The approach is compositional, and can be used to specify other system's properties such as functional and temporal requirements. The use of a common formalism eases the integration of security requirements into system requirements so that they can be reasoned about uniformly throughout the development lifecycle. Furthermore specification of policies are executable in *Tempura*, a simulation tool for ITL.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of computing and Information Systems**]: Security and Protection

General Terms

Security, Verification

Keywords

Authorization, delegation, policy composition, access control

FMSE'03, October 30, 2003, Washington, DC, USA.

1. INTRODUCTION

Recently, researchers have become increasingly interested in developing authorization models which are flexible and expressive enough so as to handle the specification and enforcement of multiple policies. In practice, a single policy is not general enough to achieve the level of protection required in many real-world applications. Rather a combination of such policies applies.

Woo and Lam [20] have proposed a general framework to specify authorization rules based on default logic. Positive and negative authorization rules can be specified in their model. However the approach provides no mechanisms to handle conflicting authorizations which might be derived using the rules. Jajodia et al. [12] have addressed this problem and provided specific rules (decision rules) to resolve conflicts among authorizations. However the validity periods of authorization rules cannot be specified nor can temporal dependencies among authorizations be expressed in their framework.

As pointed out in [4], permissions are often limited in time or may hold for specific periods of time. A temporal model for access control has been proposed by Bertino et al. [4, 3]. A time interval is associated with each authorization to determine the period of time in which the authorization holds. Temporal dependencies among authorizations can be expressed. However their framework cannot handle the enforcement of multiple policies. Neither can delegation of access rights be expressed. Ponder [8] is a high-level policy specification language in which authorization and delegation policies, among others, can be specified. But it does not support formal reasoning.

A compositional formal framework for access control which can support delegation, timing and the enforcement of multiple policies has not yet been investigated. On the other hand current access control models treat security concerns independently, and their actual incorporation into systems development lifecycle is not well understood. Similar remark has been made by Devanbu et al. [9]. We argue that security requirements must be integrated with the system requirement so that the properties of the system can be reasoned about in a uniform manner throughout the development lifecycle. We believe that this will significantly contribute to the development of secure systems.

This paper presents a compositional formal framework for the specification of access control policies. The approach can

handle the enforcement of multiple policies through policies composition. Interval Temporal Logic (ITL) [14] is our underlying formal framework. Authorization rules and delegation rules are formulated as safety properties. Schneider [19] showed that access control can be expressed as safety properties. However other security policies such as *information flow* and *availability* are not, in general, safety properties [19, 13]. Both information flow and availability are not addressed in this paper. The approach combines temporal modalities and boolean connectives for policies composition. This provides a higher degree of expressiveness and can support the specification of several protection requirements that cannot be expressed in traditional authorization models. For example, policies can be composed in sequence to express a complex policy which evolves over periods of time. Moreover policies specifications are executable in *Tempura*, a simulation tool for ITL.

ITL provides a compositional proof system based on *assumptions-commitments* paradigm for reasoning about functional and temporal properties of systems [15, 21]. As we aim to handle both security requirements as system requirements, it is useful to specify them in the same formal notation. This has motivated our choice of framework. Furthermore, a refinement calculus is provided for transforming a high-level abstract specification (written in ITL) into a more concrete specification executable in Tempura [7]. It follows that ITL can be used to describe a system at different levels of abstraction. However security requirements must be considered at the early phase of the development lifecycle.

The remainder of the paper is organised as follows. Section 2 gives an overview of ITL. The access control model is presented in Section 3. Section 4 formalises a mechanism for access control deployment. Implementation issues are addressed in Section 5 using an example. The paper ends with a discussion in Section 6.

2. INTERVAL TEMPORAL LOGIC

ITL is a linear-time temporal logic with a discrete model of time. An interval is considered to be a (in)finite, nonempty sequence of states $\sigma_0\sigma_1\ldots$, where a state σ_i is a mapping from the set of variables to the set of values (integers). The length $|\sigma|$ of a finite interval σ is equal to the number of states in the interval minus one. An empty interval has exactly one state and its length is equal to 0.

The syntax of ITL is defined as follows, where μ is a constant, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, and p is a predicate symbol.

- Expressions

$$e ::= \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a : f$$

- Formulas

$$f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \cdot f \mid \texttt{skip} \mid f_1; f_2 \mid f^*$$

The informal semantics of the most interesting constructs are stated as follows. The expression $\imath a : f$ denotes a value a such that the formula f holds. The formula `skip` denotes the unit interval (length equal to 1). The formula $f_1; f_2$ holds for an interval if the interval can be decomposed ("chopped") into a prefix and a suffix interval, such that f_1 holds over the prefix and f_2 holds over the suffix, or if the interval is infinite and f_1 holds for that interval. Finally the formula f^* holds for an interval if the interval is decomposable into finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

Note that there are three primitive temporal operators in ITL: `skip`, ";" (*chop*), and "*" (*chopstar*). A *state formula*, commonly denoted by w, is a formula with no temporal operators in it. Following are some samples of formulas with their informal meaning.

- $I = 1$ holds for an interval if I's value in the interval's initial state is equal to 1.
- $\texttt{skip}; I = 5$ holds for an interval if I's value in the interval's second state is equal to 5.
- $I = 1; I = 3$ holds for an interval if I's value in the initial state is equal to 1 and the value of I is equal to 3 in some other state (not necessary the second) of the interval.
- $\neg(true; I = 0)$ holds for an interval if I's value is never equal to 0 within the interval.

The formal semantics of expressions and formulas are summarized in Table 1 and Table 2 respectively. Readers are referred to [7, 6] for details on the logic. The semantics of a statement X over an interval σ is denoted by $[\![X]\!]_\sigma$. We also denote by χ a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v, i.e $|\sigma| = |\sigma'|$ and for all i, $0 \le i \le |\sigma|$ and $v' \neq v$, $\sigma_i(v') = \sigma'_i(v')$.

Table 1: Formal semantics of expressions

$$[\![\mu]\!]_\sigma = \mu.$$

$$[\![v]\!]_\sigma = \sigma_0(v).$$

$$[\![g(e_1, \ldots, e_n)]\!]_\sigma = g([\![e_1]\!]_\sigma, \ldots, [\![e_n]\!]_\sigma).$$

$$[\![\imath a : f]\!]_\sigma = \begin{cases} \chi(u) \text{ if } u \neq \{\} \\ \chi(a) \text{ otherwise} \end{cases}$$

where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \text{ and } [\![f]\!]_{\sigma'} = tt\}$

Obviously common temporal modalities such as $\Box$ (always) and $\Diamond$ (sometime) can be expressed. Some frequently used abbreviations are given in table 3. In particular, if f is a formula then the formula

- $\Box f$ holds for an interval if f holds for any suffix of the interval.
- $\Diamond f$ holds for an interval if f holds for some suffix of the interval.
- $\boxdot f$ holds for an interval if f holds for any subinterval of the interval.
- $\diamondsuit f$ holds for an interval if f holds for some subinterval of the interval.

Table 2: Formal semantics of formulas

$[\![p(e_1,\ldots,e_n)]\!]_\sigma = tt$ iff $p([\![e_1]\!]_\sigma,\ldots,[\![e_n]\!]_\sigma)$.

$[\![\neg f]\!]_\sigma = tt$ iff $[\![f]\!]_\sigma = ff$.

$[\![f_1 \wedge f_2]\!]_\sigma = tt$ iff $[\![f_1]\!]_\sigma = tt$ and $[\![f_2]\!]_\sigma = tt$.

$[\![\forall v.f]\!]_\sigma = tt$ iff for all σ' s.t. $\sigma \sim_v \sigma'$, $[\![f]\!]_{\sigma'} = tt$.

$[\![\texttt{skip}]\!]_\sigma = tt$ iff $|\sigma| = 1$.

$[\![f_1; f_2]\!]_\sigma = tt$ iff
(exists k, s.t. $[\![f_1]\!]_{\sigma_0\ldots\sigma_k} = tt$ and
((σ is infinite and $[\![f_2]\!]_{\sigma_k\ldots} = tt$) or
(σ is finite and $k \leq |\sigma|$ and $[\![f_2]\!]_{\sigma_k\ldots\sigma_{|\sigma|}} = tt$)))
or (σ is infinite and $[\![f_1]\!]_\sigma = tt$).

$[\![f^*]\!]_\sigma = tt$ iff
if σ is infinite then
(exist $l_0,\ldots,l_n$ s.t. $l_0 = 0$ and $[\![f]\!]_{\sigma_{l_n}\ldots} = tt$ and
for all $0 \leq i < n$, $l_i \leq l_{i+1}$ and $[\![f]\!]_{\sigma_{l_i}\ldots\sigma_{l_{i+1}}} = tt$.)
or
(exist an infinite number of l_i s.t. $l_0 = 0$ and $[\![f]\!]_{\sigma_{l_n}\ldots} = tt$ and
for all $0 \leq i < n$, $l_i \leq l_{i+1}$ and $[\![f]\!]_{\sigma_{l_i}\ldots\sigma_{l_{i+1}}} = tt$.)
else
(exist $l_0,\ldots,l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
for all $0 \leq i < n$, $l_i \leq l_{i+1}$ and $[\![f]\!]_{\sigma_{l_i}\ldots\sigma_{l_{i+1}}} = tt$.)

3. ACCESS CONTROL POLICY

Access control is a security measure that consist of ensuring that only authorized accesses can take place in a system. An access control policy specifies the access rights, that is who is authorized to access the system and what actions he is allowed to perform on the system's resources. Ensuring security is tricky and formal notations are increasingly used to specify security policies [2, 1, 5]. This section describes our formal framework for the specification of policies, based on Interval Temporal Logic presented in the previous section. We assume a finite set $\mathcal{S}$ of subjects s, a finite set $\mathcal{O}$ of objects o, and a finite set $\mathcal{A}$ of actions a. A subject denotes any entity capable to perform actions on objects. Objects denotes systems resources under protection, while actions are operations that can be performed on them. Hierarchical relationships can be defined within any of these sets to describe dependencies among their elements. For example subjects can be organized into groups or roles [18].

3.1 Authorization Model

A positive authorization is generally denoted by a triple $(s, o, +a)$ meaning that the subject s is authorized to execute the action a on the object o. On the contrary, a negative authorization denoted by $(s, o, -a)$ means that the subject s is not authorized to execute the action a on the object o. These two concepts are usually used to specify access control policies.

3.1.1 Formalising Authorization

In ITL we use two boolean arrays $autho^+$ and $autho^-$ to model positive and negative authorizations respectively. Therefore a positive authorization $(s, o, +a)$ is denoted by $autho^+(s, o, a)$, where s, o, and a are indices. This authorization holds if the value of $autho^+(s, o, a)$ equals *true* and does not hold otherwise. Similarly, $autho^-(s, o, a)$ models a negative authorization $(s, o, -a)$. Positive and negative authorizations are used at the specification level to state who is or is not allowed to do what. As we will show in the sequel, the use of signed (i.e positive/negative) authorizations gives more flexibility in handling authorization rules. For example, negation can be banned in consequences of rules without lost of generality (cf. Section 3.1.2). A signed authorization model is more expressive in that it can specify different kind of policies such as open, closed and hybrid policies (see [12] for more details).

On the other hand, a mechanism is needed to infer from the specification (based on signed authorizations) the actual access rights of each subject. A simple example is a mechanism to resolve conflicts that might occur (e.g. when $autho^+(s, o, a)$ and $autho^-(s, o, a)$ hold at the same time) within the specification. Hence, we consider another boolean array *autho* in which the final decision is taken as who is allowed to do what. The value of *autho* is computed from those of $autho^+$ and $autho^-$ used to describe the access rights. This means that *autho* can be thought of as the access control matrix which is used to enforce security. Another boolean array *error* is needed to signal errors or undesired behaviours within the specification so that the security manager can be aware and fix them timely. For example, when conflicting authorizations are not allowed, the variable *error* is used to signal any occurence of these.

A security policy must determine at any time the access rights of each subject with respect to any object and any action. Writing a complete specification to state this can be very complex and cumbersome. It is convenient to have a specification that contains only variables which

Table 3: Frequently used abbreviations

$inf \mathrel{\widehat{=}} true; false$	infinite interval	$finite \mathrel{\widehat{=}} \neg inf$	finite interval
$\bigcirc f \mathrel{\widehat{=}}$ skip;f	next f	$\bigcirc e \mathrel{\widehat{=}} \imath a : \bigcirc(e = a)$	value of e at the next state
$empty \mathrel{\widehat{=}} \neg more$	empty interval	$more \mathrel{\widehat{=}} \bigcirc true$	non-empty interval
$\Diamond f \mathrel{\widehat{=}} finite; f$	sometimes f	$\Box f \mathrel{\widehat{=}} \neg\Diamond\neg f$	always f
$\diamondsuit f \mathrel{\widehat{=}} finite; f; true$	subinterval	$\boxdot f \mathrel{\widehat{=}} \neg\diamondsuit\neg f$	all subintervals
$f_1 \supset f_2 \mathrel{\widehat{=}} \neg f_1 \vee f_2$	implies	$\boxplus f \mathrel{\widehat{=}} \Box(more \supset f)$	all mostly
$fin\ f \mathrel{\widehat{=}} \Diamond(empty \supset f)$	final state	$len \mathrel{\widehat{=}} \imath a : intlen(a)$	length of interval
if f_0 then f_1 else $f_2 \mathrel{\widehat{=}}$		$e_1 := e_2 \mathrel{\widehat{=}} (\bigcirc e_1) = e_2$	assignment
$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else	$intlen(e) \mathrel{\widehat{=}} \exists I.((I = 0) \wedge \boxplus(I := I + 1) \wedge fin(I = e))$	

are constrained to change eventually, other variables being assumed stable in the scope of the specification. This situation is known as the *frame problem* [10] and is intensively addressed in artificial intelligence settings. Although the concept of *frame variables* is a suitable solution to the problem, it is not convenient to our framework as it is not flexible enough. We will rather follow the idea of Hale [10] which uses the concept of *default values*. The intuition is that when a variable is not explicitly assigned a value, it is implicitly assigned a default value. We assume that $autho^+(s,o,a)$, $autho^-(s,o,a)$, $autho(s,o,a)$, and $error(s,o,a)$ for all s, o, and a, have default values and their default value is $false$ for security reason, viz to signal that no permission is granted and no error has occured.

3.1.2 Authorization rules

Current authorization models express a policy in terms of authorization rules. In our case, a policy can be specified as a safety formula expressing how access rights are granted over time. However the use of rules makes the specification clearer and easier to understand. In this respect we define the operator $\mapsto$ over formulas as

$$f \mapsto w \mathrel{\widehat{=}} \boxdot(f \supset \Diamond(f; w)) \qquad (1)$$

where f stands for any temporal formula, and w is a state formula. The formula $f \mapsto w$ states that any subinterval satisfying f such that f does not hold on any of its prefixes (other than itself) ends in a state satisfying w. Intuitively this means that if f holds then w must follow. If f is a state formula then the formula (1) is a (authorization) rule in the sense of [3, 12], where f is the premise and w is the consequence.

In our framework the premise can be any temporal formula. This allows the specification of complex authorization rules, such as those expressing time dependencies among authorizations. Samarati et al. in [16] suggested to attach more general conditions to authorization rules in order to specify their validity based on the system state, the state of objects or the history of authorizations. In formula (1), f can be used to express those features, for example f can be a temporal formula specifying some property on the execution history. We distinguish two kinds of rules: *signed authorization rules* and *authorization enforcement rules*. Signed authorization rules state how positive/negative authorizations are inferred as formulated in Definition 1. They are used by the security manager to specify the access rights to the system. Note that negation is not allowed in the consequences of these rules. This is not restrictive and constitutes an advantage of using signed authorizations because contradictions can thus be avoided without loss of generality.

Definition 1. (Signed authorization rule)
A signed authorization rule has one of the following forms

- $f \mapsto autho^+(s,o,a)$ (positive authorization rules)
- $f \mapsto autho^-(s,o,a)$ (negative authorization rules)

for some subject s, object o and action a, where f stands for any (temporal) formula.

Example 1 shows some samples of signed authorization rules in which $in(s_1, s_2)$ means that subject s_1 is a member of the group s_2.

Example 1.

- Permissions for a group propagate to members

$$(in(s_1,s_2) \wedge autho^+(s_2,o,a)) \mapsto autho^+(s_1,o,a)$$

- Permissions are limited in time

$$(autho^+(s,o,a) \wedge len = 5) \mapsto autho^-(s,o,a)$$

Enforcement rules are devised to specify the enforcement mechanism, i.e how the access rights are derived from positive and negative authorizations specifications. Inconsistencies amongst signed authorizations are resolved using these rules. The general form of enforcement rules is given in Definition 2.

Definition 2. (Enforcement rule)
An (authorization) enforcement rule is a formula of the form

$$f \mapsto autho(s,o,a)$$

for some subject s, object o and action a, where f stands for any (temporal) formula.

Jajodia et al. in [12] identified three main categories of policies: closed policies, open policies and hybrid policies. Closed policies allowed only positive authorization in the specifications. They are enforced by the rules

$$autho^+(s,o,a) \mapsto autho(s,o,a)$$

which means that only privileges explicitly stated are granted. Open policies in which only negative authorizations are allowed in the specifications are enforced by the rule

$$\neg autho^-(s,o,a) \mapsto autho(s,o,a).$$

Therefore privileges are granted if not explicitly denied. Hybrid policies allow positive and negative authorizations to be specified. Conflicts among authorizations in these policies can be handled using enforcement rules as follows.

- Permissions take precedence, viz.

$$autho^+(s,o,a) \mapsto autho(s,o,a)$$

- Denials take precedence, viz.

$$(autho^+(s,o,a) \wedge \neg autho^-(s,o,a)) \mapsto autho(s,o,a)$$

- No conflicts allowed, viz.

$$\begin{cases} autho^+(s,o,a) \mapsto autho(s,o,a) \\ \wedge \\ (autho^+(s,o,a) \wedge autho^-(s,o,a)) \mapsto error(s,o,a) \end{cases}$$

More sophisticated enforcement mechanisms (such as subgroups/path overriding) can be devised using enforcement rules. Another important issue in (discretionary) access control concerns the delegation of access rights which is addressed in the following section.

3.2 Delegation Model

Delegation is a mechanism which enables a subject to delegate some of its rights to another subject for it to act on its behalf. In discretionary access control delegation of rights is at the discretion of subjects. This means that the initiative to delegate is taken by subjects and not the policy manager. However it should be possible to control delegations through access control policy to ensure security, especially in systems allowing cascaded delegations. A delegation policy specifies the ability of subjects (the grantors) to delegate access rights to other subjects (the grantees) to perform actions on their behalf. Positive delegation policies grant the right to delegate while negative delegation policies forbid delegation.

Similarly as authorization policies (presented in the previous section) we model positive delegation by a boolean array $candeleg^+$ such that $candeleg^+(s_1,s_2,o,a)$ (where s_1, s_2, o, a are indices) is equal to *true* if subject s_1 can be permitted to delegate to subject s_2 the right to perform action a on object o. In the same way, the boolean array $candeleg^-$ models negative delegations, i.e $candeleg^-(s_1,s_2,o,a)$ is *true* if s_1 cannot be permitted to delegate to subject s_2 the right to perform action a on object o. We use another array *candeleg* that plays the same role as *autho* for authorizations. Thus the value of $candeleg(s_1,s_2,o,a)$ is equal to *true* if subject s_1 is permitted to delegate to subject s_2 the right to perform action a on object o, and *false* otherwise. For similar reason as for authorization policies, we assume that $candeleg^+(s_1,s_2,o,a)$, $candeleg^-(s_1,s_2,o,a)$, and $candeleg(s_1,s_2,o,a)$ for all s_1, s_2, o, a, have default values and their default value is *false*. Signed delegation rules and delegation enforcement rules are defined in the same way as for authorizations. Errors are handled in the similar way as well.

However, a subject should also be able to revoke a right it has delegated to another. Moreover a delegated right should be automatically revoked if the grantor loses that right. This mechanism is formulated by the following rule

$$\left(\begin{array}{c} autho(s_1,o,a) \\ \wedge \\ candeleg(s_1,s_2,o,a) \\ \wedge \\ deleg(s_1,s_2,o,a) \end{array} \right) \mapsto autho(s_2,o,a) \qquad (2)$$

where *deleg* is a boolean array used as follows. The element $deleg(s_1,s_2,o,a)$ of the array is used by the grantor s_1 to delegate to (by setting the variable to true) and revoke from (by setting the variable to false) the grantee s_2 the right to perform action a on object o. This rule is added to any policy which allows delegation.

The following section describes simple policies which are merely conjunction of rules, and some mechanisms for manipulating them.

3.3 Simple Policy

In practice, security policies are not static but rather evolve continuously to fix new security breaches or to meet new security requirements. This dynamics might lead to some rules being withdrawn from and new ones added to the policy. An authorization model must provide a mechanism to ease the task of the security manager on these matters. In this section we show how rules can be activated/deactivated and how new rules can be added to policies.

The general form of a simple policy is defined as

$$P \mathrel{\widehat{=}} w \wedge \ \wedge_{i \in I} R_i \ \wedge fin\ w'$$

where P stands for policy, w is a state formula that holds for the initial state of any interval on which the policy holds, R_i, $i \in I$ are (authorization/delegation) rules and I a finite set of natural numbers, and w' is a state formula that holds for the final state of any interval on which the policy holds. The intuition is that the rules ensure security in any interval satisfying P, while the state formulas w and w' control its boundaries. In the sequel we denote by $\overleftarrow{P}$ and $\overrightarrow{P}$, respectively the initial state w and the final state w' of a policy P.

3.3.1 Adding a rule to a policy

A rule R can be added to a policy P by simple conjunction to form a new policy

$$P \wedge R$$

which enforces both P and R. This provides a way for incremental development of a security policy. Note that R cannot clash with other rules in P since negation is not allowed in the consequences of rules.

3.3.2 Activation/deactivation of rules within policy

A mechanism to activate/deactivate rules within a policy might consist of adding a flag (which is a state formula) in the premises of rules, viz.

$$R_i \mathrel{\widehat{=}} (f_{i1} \wedge flag_i) \mapsto f_{i2}.$$

The rule is "activated" when the flag holds and "deactivated" otherwise. For example to deactivate the rule R_j in a policy P, we just take the conjunction of P with the formula $\Box(\neg flag_j)$, viz.

$$P \wedge \Box(\neg flag_j).$$

This feature can be generalized to the configuration of policies.

3.3.3 *Configuration of policy*

A configuration is a mechanism that allows us to determine (dynamically) which rules apply and which ones do not within a policy. We call a *configuration rule* a rule of one of the following forms

- $f \mapsto flag$ (activation)
- $f \mapsto \neg flag$ (deactivation)

where f stands for any formula and $flag$ is a state formula.

A configuration C is then a conjunction of configuration rules c_i, $i \in I$, for some finite set of indices I, viz.

$$C \mathrel{\widehat{=}} \wedge_{i \in I}\, c_i.$$

If P is a policy and C a configuration then $P \wedge C$ is the policy obtained by configuring P with C.

Complex policies are devised by policies composition as discussed in the following section.

3.4 Policy Composition

Policies are closed under the following operators. Let P_1, P_2 and P stand for policies, and w for a state formula. The initial state $\overleftarrow{P}$ and the final state $\overrightarrow{P}$ of a compound policy are defined inductively in Table 4.

Table 4: Initial and final states

$\overleftarrow{P_1; P_2} = \overleftarrow{P_1}$	$\overrightarrow{P_1; P_2} = \overrightarrow{P_2}$
$\overleftarrow{P_1 \vee P_2} = \overleftarrow{P_1} \vee \overleftarrow{P_2}$	$\overrightarrow{P_1 \vee P_2} = \overrightarrow{P_1} \vee \overrightarrow{P_2}$
$\overleftarrow{P_1 \wedge P_2} = \overleftarrow{P_1} \wedge \overleftarrow{P_2}$	$\overrightarrow{P_1 \wedge P_2} = \overrightarrow{P_1} \wedge \overrightarrow{P_2}$
$\overleftarrow{P^+} = \overleftarrow{P}$	$\overrightarrow{P^+} = \overrightarrow{P}$
$\overleftarrow{\text{if } w \text{ then } P_1 \text{ else } P_2} =$ if w then $\overleftarrow{P_1}$ else $\overleftarrow{P_2}$	$\overrightarrow{\text{if } w \text{ then } P_1 \text{ else } P_2} = \overrightarrow{P_1} \vee \overrightarrow{P_2}$

3.4.1 *Parallel*

The parallel composition of P_1 and P_2 is the policy

$$P_1 \wedge P_2$$

which holds if both P_1 and P_2 hold. A system which enforces the policy $P_1 \wedge P_2$ enforces both the policies P_1 and P_2 simultaneously.

3.4.2 *Sequence*

Two policies P_1 and P_2 can be composed in sequence to form the policy

$$P_1; P_2$$

which behaves like P_1 for some time, then like P_2 afterwards, provided that $\overrightarrow{P_1} \supset \overleftarrow{P_2}$. That is P_1 and P_2 must agree at the transition state where the behavior related to P_1 ends and that related to P_2 commences. In general, organisations apply different policies for specific periods of time. Universities distinguish between terms time and vacations. Banks render restricted services in the week-end and holidays if they are not merely closed. Traditional authorization models cannot express sequential composition of policies.

3.4.3 *Conditional*

In certain conditions one policy say P_1, might apply and not the other say P_2. The conditional allows to express such a policy as

$$if\ w\ then\ P_1\ else\ P_2$$

where the guard w determines which of P_1 and P_2 applies. When w holds P_1 applies otherwise P_2 applies. A typical example, in any organisation, might be P_1 applies for staff and P_2 for non-staff. Note that P_1 and/or P_2 might also be conditionals, refining staff and non-staff further into different subcategories.

3.4.4 *Disjunction*

By contrast to conditional, disjunction specifies a policy denoted by

$$P_1 \vee P_2$$

which non-deterministically behaves like P_1 or like P_2. The choice between P_1 and P_2 is made internally at run-time by the system, and cannot be pre-computed.

3.4.5 *Iteration*

In some organisations, a policy say P, is adopted for a given period of time (days, weeks, months, etc), then repeated successively over consecutive periods. Such a protection requirement can be formulated as

$$P^+$$

(i.e $P; P^*$) meaning that the policy P is iterated over non-empty finite sequence of periods of time, provided that $\overrightarrow{P} \supset \overleftarrow{P}$.

3.4.6 *Scope*

The scope of a policy say P, can be limited in time, i.e

$$P \wedge len = d$$

for some duration d.

4. ACCESS CONTROL DEPLOYMENT

Access control can be thought of as a security component whose role is to control access to data. Such a component must be able to handle all the requests from subjects (users) in accessing information and check their access rights w.r.t the security policy that applies. In this section we formalise the security requirement for access control and investigate its implementation in Tempura.

4.1 Security Requirement for Access Control

The requirement of access control is well understood and states that *only authorized parties can access the system's resources.* This is formulated in our model as

$$req \mathrel{\widehat{=}} \Box(access(s, a, o) \supset autho(s, a, o)) \qquad (3)$$

where $access(s, a, o)$ is equal to true if subject s has performed action a on object o. The authorization matrix *autho* is set by the access control policy, as described in the previous section. Instead, the array *access* is set by the reference monitor in response to access requests.

4.2 Implementation

A simplified model for access control deployment is depicted in Figure 1. The reference monitor is responsible for enforcing security. All the subjects' requests for accessing data are sent to the reference monitor, which authenticates the senders and checks their access permissions. If a sender has the required access right for its request, then its request is successful otherwise it is rejected.

The behaviour of the reference monitor is formalised as

$$rm \mathrel{\widehat{=}} \Box \left(\begin{array}{c} \forall s, a, o.\ \text{if } request(s, a, o) \wedge autho(s, a, o) \\ \text{then } access(s, a, o) = true \\ \text{else } access(s, a, o) = false \end{array} \right) \tag{4}$$

where *request* is a boolean array set by subjects so that $request(s, a, o)$ is equal to true if subject s requests to perform action a on object o. Authentication protocols are not addressed in this paper.

Lemma 1 states that the reference monitor sets the value of $access(s, a, o)$ to $false$ if subject s does not request or is not allowed to perform action a on object o.

LEMMA 1.

$$\vdash rm \supset \Box((\neg request(s, o, a) \vee \neg autho(s, o, a)) \supset \neg access(s, o, a))$$

PROOF. The proof of Lemma 1 is straightforward from the definition of rm (formula (4)). □

Similarly the behaviour of subjects w.r.t their requests in accessing objects can be formulated as an ITL formula *users*. Let *policy* denotes the access control policy. The following theorem states the correctness of the design.

THEOREM 1.

$$\vdash (users \wedge rm \wedge policy) \supset req$$

PROOF. The proof is carried out by contradiction. So suppose that $users \wedge rm \wedge policy \wedge \neg req$ holds for some interval σ. It follows that σ satisfies both formulas rm and $\Diamond(access(s, o, a) \wedge \neg autho(s, o, a))$. From Lemma 1, this leads to a contradiction. □

This access control mechanism is executable in *Tempura* as we show in the following example.

5. AN EXAMPLE

We consider an institution in which the process of making exams is fully controlled by a secure computer system. The exams are stored on a server which controls the access to them. Each exam is assigned an examiner, a moderator and an external examiner. The process comprises seven phases and each phase i lasts at most d_i time units (say days), and covers a specific task as follows.

Phase 1 The examiner prepares the first draft of the exam. In this phase he is the only one allowed to access the exam.

Phase 2 The moderator can access the exam and comment it. The examiner cannot access the exam in this phase, neither can the external examiner and students.

Phase 3 The examiner can access the exam to revise it w.r.t the moderator's comments and suggestions. No one else is allowed to access it.

Phase 4 It is the turn to the external examiner to assess the exam. So he accesses the exam and comments it. However he is the only one to access the exam in this phase.

Phase 5 The examiner can access the exam for the final revision.

Phase 6 The final release is kept securely. No one can access it until the time the exam takes place.

Phase 7 Exam period. Students who attend the exam can read it. So can the examiner, the moderator and the external examiner. No write access is allowed.

5.1 Signed authorization rules

Following authorization rules apply.

- Examiner can read:

$$examiner\_r^{+} \mathrel{\widehat{=}} examiner(s, o) \mapsto autho^{+}(s, o, read)$$

- Examiner can write:

$$examiner\_w^{+} \mathrel{\widehat{=}} examiner(s, o) \mapsto autho^{+}(s, o, write)$$

- Examiner cannot read:

$$examiner\_r^{-} \mathrel{\widehat{=}} examiner(s, o) \mapsto autho^{-}(s, o, read)$$

- Examiner cannot write:

$$examiner\_w^{-} \mathrel{\widehat{=}} examiner(s, o) \mapsto autho^{-}(s, o, write)$$

where $examiner(s, o)$ means that s is the examiner assigned to the exam o.

In the sequel the abbreviations

$$examiner^{+} \mathrel{\widehat{=}} examiner\_r^{+} \wedge examiner\_w^{+}$$

and

$$examiner^{-} \mathrel{\widehat{=}} examiner\_r^{-} \wedge examiner\_w^{-}$$

will be used for simplicity. Similar rules can be defined for moderators, external examiners, and students.

5.2 Enforcement rule

Each phase can apply specific enforcement mechanism. Yet for simplicity, we will consider a single conflict resolution mechanism that denials take precedence in the event of conflict, viz.

$$denialsTakePrec \mathrel{\widehat{=}} ((autho^{+}(s, o, a) \wedge \neg autho^{-}(s, o, a)) \mapsto autho(s, o, a))$$

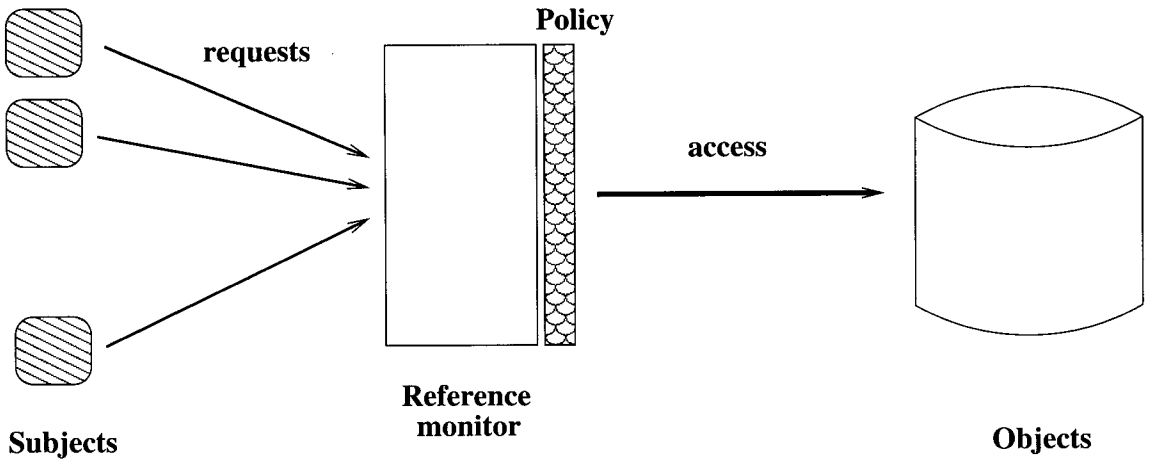

Figure 1: A model for access control deployment

5.3 Policy

As described above, each phase enforces specific access control policy. Thus the policy for the system can be expressed as the sequential composition of the policies enforced in the phases. These policies are defined in Table 5 such that p_1 is enforced in phases 1, 3, and 5, while p_2, p_3, p_4, and p_5 apply respectively in phase 2, 4, 6 and 7. Then the access control policy for the exam system is

$$policy \mathrel{\widehat{=}} \left(\begin{array}{c} (len = d_1 \wedge p_1); \text{skip}; (len = d_2 \wedge p_2); \\ \text{skip}; (len = d_3 \wedge p_1); \text{skip}; (len = d_4 \wedge p_3); \\ \text{skip}; (len = d_5 \wedge p_1); \text{skip}; (len = d_6 \wedge p_4); \\ \text{skip}; (len = d_7 \wedge p_5) \end{array} \right)$$

5.4 Simulation

We will consider the following scenario, describing the behaviour of users who request to access the exams. To simplify the discussion, we have just one examiner (*Bob*), one moderator (*Alice*), one external examiner (*Dave*), and one student (*Carol*). Each phase lasts 2 time units. The users behave as follows, where T stands for time.

- *Bob* requests to write the exam in phases 1, 3, 4, and 5. That is

  $$bobReq \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \text{if } (T = 0 \vee T = 7 \vee T = 9 \vee T = 13) \\ \text{then } request(bob, exam, write) \end{array} \right)$$

  However, as an examiner he is not allowed to access the exam in phase 4.

- *Alice* requests to read the exam in phase 1 and to write it in phases 2 and 5. That is

  $$aliceReq \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \text{if } T = 2 \\ \text{then } request(alice, exam, read) \\ \wedge \\ \text{if } (T = 3 \vee T = 13) \\ \text{then } request(alice, exam, write) \end{array} \right)$$

  However, as a moderator, she is not allowed to access the exam in phase 1.

- *Dave* requests to write the exam in phases 4 and 7. That is

  $$daveReq \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \text{if } (T = 10 \vee T = 18) \\ \text{then } request(dave, exam, write) \end{array} \right)$$

- *Carol* is a student and requests to read the exam in phases 6 and 7. That is

  $$carolReq \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \text{if } (T = 15 \vee T = 18) \\ \text{then } request(carol, exam, read) \end{array} \right)$$

Let

$$users \mathrel{\widehat{=}} bobReq \wedge aliceReq \wedge daveReq \wedge carolReq.$$

Then the simulation of the formula

$$users \wedge rm \wedge policy$$

in Tempura gives the following monitoring of the requests in accessing the exam.

```
State   0: ------------------------------------------
State   0: Subjects   Objects      Actions  Request ...
State   0: ------------------------------------------
State   0: Bob        Exam         write    accepted
State   2: Alice      Exam         read     rejected
State   3: Alice      Exam         write    accepted
State   7: Bob        Exam         write    accepted
State   9: Bob        Exam         write    rejected
State  10: Dave       Exam         write    accepted
State  13: Bob        Exam         write    accepted
State  13: Alice      Exam         write    rejected
State  15: Carol      Exam         read     rejected
State  18: Dave       Exam         write    rejected
State  18: Carol      Exam         read     accepted

Done!  Computation length:  20.  Total Passes:  24.
Total reductions: 47170 (46706 successful). ...
```

The access control matrix can also be visualized and checked w.r.t the policy specification. For example this is the access control matrix in state 0.

```
State   0: ------------------------------------------------
State   0: Subjects   Objects      Actions  Authorization
State   0: ------------------------------------------------
State   0: Bob        Exam         read     permitted
State   0: Bob        Exam         write    permitted
State   0: Alice      Exam         read     denied
State   0: Alice      Exam         write    denied
State   0: Dave       Exam         read     denied
State   0: Dave       Exam         write    denied
State   0: Carol      Exam         read     denied
State   0: Carol      Exam         write    denied
State   1: Bob        Exam         read     permitted
...
```

Table 5: Policy for each phase

$$p_1 \;\widehat{=}\; \begin{pmatrix} examiner^+ \wedge \\ moderator^- \wedge \\ external^- \wedge \\ student^- \wedge \\ denialsTakePrec \end{pmatrix}, \quad p_2 \;\widehat{=}\; \begin{pmatrix} examiner^- \wedge \\ moderator^+ \wedge \\ external^- \wedge \\ student^- \wedge \\ denialsTakePrec \end{pmatrix},$$

$$p_3 \;\widehat{=}\; \begin{pmatrix} examiner^- \wedge \\ moderator^- \wedge \\ external^+ \wedge \\ student^- \wedge \\ denialsTakePrec \end{pmatrix}, \quad p_4 \;\widehat{=}\; \begin{pmatrix} examiner^- \wedge \\ moderator^- \wedge \\ external^- \wedge \\ student^- \wedge \\ denialsTakePrec \end{pmatrix},$$

$$p_5 \;\widehat{=}\; \begin{pmatrix} examiner\_r^+ \wedge examiner\_w^- \wedge \\ moderator\_r^+ \wedge moderator\_w^- \wedge \\ external\_r^+ \wedge external\_w^- \wedge \\ student\_r^+ \wedge student\_w^- \wedge \\ denialsTakePrec \end{pmatrix}.$$

6. DISCUSSION

We have developed a compositional framework for the specification of access control policies using ITL. Authorization and delegation rules are formulated as ITL formulas. The use of positive/negative authorizations and positive/negative delegations made it possible to avoid negation in the consequences of rules without loss of generality. As a result, contradiction cannot be derived from rules. While signed rules are used for the specification of access rights, enforcement rules are provided for the specification of policy enforcement mechanisms. Conflicts between positive and negative authorizations/delegations can be resolved using enforcement rules.

Simple policies are defined as conjunction of rules and additional elements to ensure security at the beginning (when the policy begins) and the end (when it terminates). Rules can be added, activated/deactivated to policies. However we do not allow dynamic creation/deletion of subjects, objects or actions. Dealing with such dynamics is known to be difficult [11, 13]. Then complex policies are devised by composition using several operators such as *chop* or *chopstar*, which are not supported by traditional approaches. Multiple policies can so be enforced through composition, and their properties reasoned about. Furthermore, specifications of policies are executable in Tempura. Since ITL can be used to reason about functional and temporal properties of systems, our approach provides a uniform formal framework to incorporate security policy specifications and system requirements. We recommend to consider security concerns at the early stage of the development lifecycle.

6.1 Run-time Verification

We aim to develop technologies and tool support for the continual enforcement of security policies. Such a support will be based on our **SANTA** Workbench depicted in Figure 2. Currently, SANTA consists of two major agents: Monitor and Validator, where the later validates safety properties expressed in an executable subset of ITL. We propose to (a) conservatively extend ITL to provide a rich notation for expressing a variety of security policies, and (b) enhance the Validator to cater for security policies. The Monitor agent controls the information flow between the system and the Validator. The tool and model will be evaluated on a variety of case studies drawn from both military (e.g. Security policies related to NGOs) and civilian (Finance and Retailing) domains.

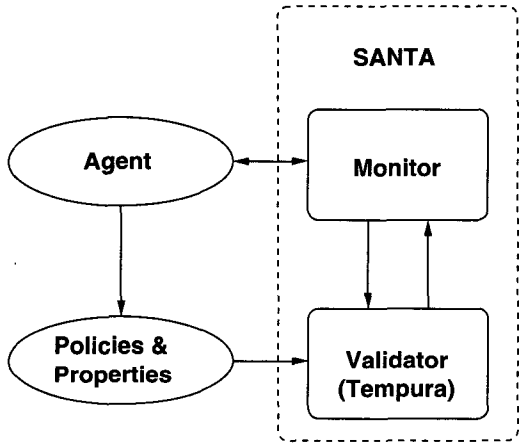

Figure 2: Architecture of SANTA.

6.2 Compositionality

Compositionality is a desirable attribute for any formal method. It allows to decompose a large system into more manageable pieces and to prove the correctness of the whole system from that of its immediate components. In addition it supports early reasoning of designs, i.e one can reason with the specification of the components without knowing their implementation.

In ITL, compositionality is provided through *assumptions-commitments* paradigm whose general form is illustrated as follows.

$$\vdash w \wedge As \wedge Sys \supset Co \wedge fin\ w'.$$

This states that if the (state) formula w is true in the initial state and the assumption As holds over the interval in which the system Sys is operating, then the commitment Co will hold over the interval and the (state) formula w' is true in the interval's final state or is vacuously true if the interval is infinite.

In general, the assumption As and the commitment Co can be arbitrary temporal formulas. However, special forms of these suit better for reasoning about sequential or parallel composition of systems. For sequential composition, for example, it is useful to require that As and Co be respectively fixpoints of the ITL operator $\boxdot$ (read *box-a*) and * (read *chop-star*), viz.

$$As \equiv \boxdot As, \qquad Co \equiv Co^*.$$

The first equivalence ensures that if the assumption As is true on an interval, it is also true in all subintervals. The second ensures that if zero or more sequential instances of the commitment Co span an interval, Co is also true on the interval itself. For assumption and commitment obeying the above, the following proof rule is sound.

$$\frac{\vdash w \wedge As \wedge Sys \supset Co \wedge fin\ w' \quad \vdash w' \wedge As \wedge Sys' \supset Co \wedge fin\ w''}{\vdash w \wedge As \wedge Sys; Sys' \supset Co \wedge fin\ w''.}$$

Here is an analogous rule for decomposing a proof for zero or more iterations of a formula Sys:

$$\frac{\vdash w \wedge As \wedge Sys \supset Co \wedge fin\ w}{\vdash w \wedge As \wedge Sys^* \supset Co \wedge fin\ w.}$$

Readers are referred to [15] for further details.

6.3 Related Work

Over the years, researchers have proposed a vast variety of access control policies and models [1, 2, 17, 18, 5]. A general formalism for expressing authorization rules has been proposed by Woo and Lam [20]. Their framework is based on default logic and provides interesting properties such as non-monotonicity of authorizations. That is, if a set of authorization rules is augmented by a new rule, a subject who was previously allowed access to an object may no longer be allowed the same access. Such a property can be expressed easily using defaults. However default rules might not be conclusive. As a consequence the model can lead to a situation in which an authorization request has no answer. Another limitation of their framework is the management of conflicts among authorizations, that is handled in the semantics of the authorization language.

We use the concept of default value which is conclusive compared to default rules. The idea is that when a variable is not explicitly assigned a value, it implicitly takes the default value. For example the default value for $autho^-(s, o, a)$ is $false$ meaning that if the specification does not set its value to $true$ (to say explicitly that s is denied the right to perform action a on object o) then its value is set to $false$ (to say that s is not explicitly denied the right to perform action a on object o). In addition conflicts among signed authorizations/delegations are resolved through policies enforcement mechanisms which are specified using enforcement rules.

Jajodia et al. [12] have proposed an access control model in which inconsistencies among authorizations can be resolved using rules. Their approach is more flexible and several design decisions can be chosen to handle conflicts. We follow a similar approach but provide additional mechanisms to handle delegation which cannot be specified in their framework. Their model provides a library of policies (called *FAM library*) from which policies can be extracted and enforced concurrently. However the management of the FAM library is not well understood. Moreover policies composition cannot be expressed in their authorization language. By contrast, our framework allows the enforcement of multiple policies through policies composition. This provides a way of specifying complex policies and to reason about their properties. Additionally their approach cannot specify temporal dependancies among authorizations.

Bertino et al. [3, 4] have developed a temporal model for access control. In their model time intervals are associated with authorizations to determine their validity periods. Rules are expressed using temporal relationships between authorizations. However the approach cannot handle the enforcement of multiple policies. Neither can delegation of authorization be formulated. Our framework can handle delegation and policies composition while allowing temporal reasoning about policies. Delegation is largely addressed in Ponder [8], a high-level specification language for policies management. However, as mentioned in the introduction Ponder does not support formal reasoning.

7. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(3):1–29, September 1993.

[2] D. Bell and L. Lapadula. Secure computer system unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, Bedford, MA, 1975.

[3] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A Temporal Access Control Mechanism for Database systems. *IEEE Transactions on knowledge and data engineering*, 8(1):67–80, February 1996.

[4] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transaction on Database Systems*, 23(3):213–285, September 1998.

[5] P. Bonatti, S. Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transaction on Information and System security*, 5(1):1–35, February 2002.

[6] A. Cau, B. Moszkowski, and H. Zedan. *Interval Temporal Logic.* http://www.cse.dmu.ac.uk/~cau/itlhomepage/.

[7] A. Cau and H. Zedan. Refining Interval Temporal Logic Specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, volume 1231 of *LNCS*, pages 79–94, AMAST, 1997. Sprinrg-Verlag.

[8] N. C. Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis,

Imperial College of Science, Technology and Medicine, University of London, February 2002.

[9] P. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 225–239. ACM Press, 2000. Special Volume (ICSE 2000).

[10] R. W. S. Hale. *Programming in Temporal Logic.* PhD thesis, Trinity College, University of Cambridge, October 1988.

[11] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[12] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. *ACM transaction on Database Systems*, 26(2):214–260, June 2001.

[13] J. McLean. Security Models. In J. Marciniak, editor, *Encyclopedia of Software Engineering.* Wiley Press, 1994.

[14] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, England, 1986.

[15] B. Moszkowski. Compositional reasoning using interval temporal logic and tempura. In W.-P. d. Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference*, volume 1486 of *LNCS*, pages 439–464, Berlin, 1998. Springer Verlag.

[16] P. Samarati and S. Vimercati. Access Control: Policies, Models, and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196. Springer-Verlag, September 2000.

[17] R. Sandhu. Transaction Control Expressions for Separation of Duties. *IEEE*, pages 282–286, 1988.

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[19] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[20] T. Y. C. Woo and S. S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.

[21] H. Zedan, A. Cau, and B. Moszkowski. Compositional Modelling: The Formal Perspective. In D. Bustard, editor, *Workshop on Systems Modelling for Business Process Improvement*, pages 333–354. Artech House, 2000.