Journal of Universal Computer Science, vol. 13, no. 8 (2007), 1073-1093
submitted: 22/2/07, accepted: 23/4/07, appeared: 28/8/07

Specification and Refinement of Access Control

Dominique Méry
(Nancy University & LORIA, Nancy, France
Dominique.Mery@loria.fr)

Stephan Merz
(INRIA Nancy & LORIA, Nancy, France
Stephan.Merz@loria.fr)

Abstract: We consider the extension of fair event system specifications by concepts of access control (prohibitions, user rights, and obligations). We give proof rules for verifying that an access control policy is correctly implemented in a system, and consider preservation of access control by refinement of event systems. Prohibitions and obligations are expressed as properties of traces and are preserved by standard refinement notions of event systems. Preservation of user rights is not guaranteed by construction; we propose to combine implementation-level user rights and obligations to implement high-level user rights.
Key Words: access control, event systems, refinement
Category: D.2.4 (Software/Program Verification), F.3.1 (Specifying and Verifying and Reasoning about Programs)

1 Introduction

Information systems process ever more sensitive data. They should therefore not only be correct with respect to their functional specification, but most also ensure properties related to information security. For example, data should be accessed only by (or on behalf of) users to the extent that they are authorized to do so, legal or procedural obligations must be enforced, and user rights should be respected by the system implementation.

In this paper, we propose to associate descriptions of *access control* policies with the specification of event systems. We give proof rules that allow us to verify that an event system correctly implements access control annotations, and we study to what extent such properties can be preserved when systems are refined in a framework of stepwise system development. One should note, however, that access control policies are only a part of the requirements needed for describing and assessing the security of information infrastructure. In particular, access control specifications do not constrain the use of data obtained by an authorized user. Hence, they cannot guarantee properties about information flow [McLean 1992], such as confidentiality of information.

Several formalisms for describing access control policies have been proposed in the literature. Among the more advanced ones are RBAC (Role-Based Access Control [Sandhu et al. 1996]) and OrBAC (Organization-Based Access Control [Abou et al. 2003]). These are declarative languages that focus on the static structure of information systems and their operations. They identify the actors (abstractly represented as roles), the entities of information (abstracted as views) stored or processed by the system, and the activities by which information is accesed. Central to an access control policy is the description of constraints that state when activities are permitted or forbidden. Certain formalisms also encompass more advanced security properties such as rights or obligations. OrBAC makes a step toward specifying access control policies that may depend on run-time information by associating rights with contexts. For example, an employee's permissions could be different during working hours and off-hours. However, the effect of the activities on the system state is not described by the access-control specification, and verifying that a system implements an access-control policy is out of the scope of these formalisms.

We propose here to relate the specification of access control policies to formal models of dynamic system behavior, and we give proof rules by which one can demonstrate that a system implements an access control policy. We describe information systems within the well-known paradigm of event systems in the style of [Abrial 1996, Abrial 2003, Back and von Wright 1998]. Correctness properties of event systems can be specified as formulas of temporal logic, and there are well-established verification rules to derive properties of event systems. We are thus led to interpret access control primitives as properties of runs of event systems. Prohibitions are easily expressed as constraints on the enabling condition of events. Dually, the right of an actor to perform a certain activity can be expressed as an elementary branching-time property. The formal representation of obligations is less obvious; we propose to interpret them as liveness properties, expressible in (linear-time) temporal logic.

Beyond describing a system at a single level of abstraction, it is interesting to develop systems in a refinement-based approach. We consider in what sense access control annotations can be preserved when a system is refined. Because prohibitions and obligations are interpreted as safety and liveness properties of runs, it is not hard to see that they are indeed preserved by standard refinement conditions. However, refinement does not generally preserve branching properties, and preservation of user rights requires extra conditions. Their precise formulation is non-trivial when the "grain of atomicity" of a system description may change during refinement. We propose a condition that relies on a combination of a user right to initiate a sequence of activities (at the implementation level), and an obligation on the system to ensure that this sequence terminates, simulating the high-level activitiy. We illustrate our approach with a running example of a simple loan management system on which different access control requirements are imposed.

Related work. The existing literature on formalisms for the specification of access control considers mainly static methods of analysis. [Bertino et al. 2003] and [Cuppens et al. 2005], among others, analyze security policies for inconsistencies, and [Benferhat et al. 2003] consider techniques to resolve such inconsistencies based on stratification of rules.

Closer to our concerns are [Guelev et al. 2004, Zhang et al. 2005] who describe the use of model checking for verifying access-control policies. However, we work in a deductive framework, and we are mainly interested in verifying refinement relationships. [Koch et al. 2005] suggest a UML notation for specifying access control, together with a semantics based on graph transformation and corresponding analysis techniques. More distantly related is the work around UMLSec [Jürjens 2004], which is mainly concerned with secrecy properties.

2 Event Systems With Fairness Constraints

We describe systems as (possibly infinite-state) transition systems. More precisely, the system state is an assignment of values to a finite set of state variables, and the evolution of the system is specified by a finite set of events, in a style similar to many conventional formalisms for system specification [Abrial 1996, Back and von Wright 1998, Manna and Pnueli 1992]. Fairness conditions specify which events are assumed to occur eventually, and system properties can be expressed in linear-time temporal logic. We assume an underlying first-order interpretation whose universe is denumerably infinite. For our examples, we employ a language that contains a simple theory of (finite or denumerable) sets; in particular, functions are considered as sets of pairs $x \mapsto y$, and $\emptyset$ represents the function with empty domain. This section briefly introduces the systems that we consider and their properties, and presents rules for system verification.

2.1 Event Systems and Their Runs

Syntax of event system specifications. The specification of an event system appears in Fig. 1; it will serve as a running example[1]. The clause **constants** lists the constant parameters of a system specification, including unspecified sets whose elements appear as values of system variables. The clause **assumption** introduces a predicate that constraints the constant system parameters.

A system state is represented by state variables, introduced by the keyword **variables**; the clause **invariant** specifies a predicate Inv that the variables must satisfy at any time during system execution. Also, the clause **initial** defines a state predicate that constrains the initial states of the system.

[1] We adopt the convention [Lamport 1994b] of writing multi-line conjunctions and disjunctions as "lists" bulleted with $\wedge$ and $\vee$, relying on indentation to save parentheses.

system *Bank*
constants $Client, Loan, maxDebt$
assumption $Client \neq \emptyset \wedge Loan \neq \emptyset \wedge maxDebt \in [Client \rightarrow \mathbb{Q}]$

variables $clt, loans, due, rate, maxExtra, extra$
invariant $\wedge\ loans \subseteq Loan$
$\wedge\ clt \in [loans \rightarrow Client] \wedge due \in [loans \rightarrow \mathbb{Q}] \wedge rate \in [loans \rightarrow \mathbb{Q}]$
$\wedge\ maxExtra \in [loans \rightarrow \mathbb{Q}] \wedge extra \in [loans \rightarrow \mathbb{Q}]$
$\wedge\ \forall c \in Client : \left(\sum\{due(ll) : ll \in loans \wedge clt(ll) = c\}\right) \leq maxDebt(c)$
initial $loans = \emptyset \wedge clt = \emptyset \wedge due = \emptyset \wedge rate = \emptyset \wedge maxExtra = \emptyset \wedge extra = \emptyset$

event $newLoan(c, l, amt, dur, mx) =$
$\wedge\ c \in Client \wedge l \in Loan \setminus loans \wedge amt \in \mathbb{Q} \wedge dur \in \mathbb{N}$
$\wedge\ amt + \left(\sum\{due(ll) : ll \in loans \wedge clt(ll) = c\}\right) \leq maxDebt(c)$
$\wedge\ loans' = loans \cup \{l\} \wedge clt' = clt \cup \{l \mapsto c\}$
$\wedge\ due' = due \cup \{l \mapsto sum\} \wedge rate' = rate \cup \{l \mapsto sum/dur\}$
$\wedge\ maxExtra' = maxExtra \cup \{l \mapsto mx\} \wedge extra' = extra \cup \{l \mapsto 0\}$
fairness false

event $makePayment(l) =$
$\wedge\ l \in loans$
$\wedge\ due' = due \oplus \{l \mapsto due(l) - rate(l)\}$
$\wedge$ UNCHANGED $loans, clt, rate, maxExtra, extra$
fairness $l \in loans \wedge due(l) > 0$

event $extraPayment(l, amt) =$
$\wedge\ l \in loans \wedge amt \in \mathbb{Q}$
$\wedge\ due' = due \oplus \{l \mapsto due(l) - amt\} \wedge extra' = extra \oplus \{l \mapsto extra(l) + amt\}$
$\wedge$ UNCHANGED $loans, clt, rate, maxExtra$
fairness false
end system

Figure 1: Sample system specification.

The system transitions are described by events; we define an event e with parameters x in the form

$$\textbf{event } e(x) \triangleq BA_e(x)$$

where $BA_e(x)$ is the *before-after predicate* for the event e. Syntactically, $BA_e(x)$ is a first-order formula built from the constants declared for the system specification, the event's parameters x, as well as primed and unprimed occurrences of the system variables. Following common usage, an unprimed occurrence v of a variable denotes the value of that variable at the state before the transition, and a primed occurrence v' denotes the value at the state after the transition. More generally, for an expression t containing no primed variables, we write t' for the expression obtained by replacing every unprimed occurrence v of a variable in t by a primed occurrence v'. For expressions $t_1, \ldots, t_n$ we write UNCHANGED $t_1, \ldots, t_n$ to abbreviate the formula $t'_1 = t_1 \wedge \ldots \wedge t'_n = t_n$.

For an event $e(x)$ with before-after predicate $BA_e(x)$, we define its feasibility condition

$$\mathbf{fis}\, e(x) \;\triangleq\; \exists var' : BA_e(x) \tag{1}$$

by existentially quantifying over the primed occurrences of all state variables. Semantically, the predicate $\mathbf{fis}\, e(x)$ characterizes those states in which the event $e(x)$ can occur. For example,

$$\begin{array}{l} \mathbf{fis}\, newLoan(c, l, amt, dur, mx) \;\equiv \\ \quad \wedge\; c \in Client \wedge l \in Loan \setminus loans \wedge amt \in \mathbb{Q} \wedge dur \in \mathbb{N} \\ \quad \wedge\; amt + \left(\sum\{due(ll) : ll \in loans \wedge clt(ll) = c\}\right) \leq maxDebt(c) \end{array}$$

holds by laws of first-order logic: beyond type correctness, the feasibility condition asserts that l should be a fresh identifier for the new loan, and that the client should not become over-indebted by taking up the loan.

Fairness assumptions are associated with events by annotating the definition of an event $e(x)$ with a clause of the form

$$\mathbf{fairness}\; fair_e(x)$$

where $fair_e(x)$ is a predicate built from the constants and (unprimed) system variables, and the event parameters. The fairness assumption rules out runs of the system where $fair_e(x)$ remains true forever without $e(x)$ ever occurring. The standard condition of "weak fairness" or "justice" used in [Lamport 1994a, Manna and Pnueli 1992] is obtained when $fair_e(x)$ is the feasibility condition $\mathbf{fis}\, e(x)$. Weaker fairness hypotheses can be specified if $fair_e(x)$ is stronger than the feasibility condition. As an extreme case, no fairness is required of event $e(x)$ when $fair_e(x)$ is chosen as **false**. The framework can easily be extended to allow for strong fairness, although the language for stating properties and the associated verification rules (see Sect. 2.2) should then be extended to full temporal logic.

Well-formedness of specifications. We require certain healthiness conditions of a specification of an event system. First, the invariant asserted of a system should be implied by the initial condition and preserved by all (instances of) events to ensure that the invariant is indeed inductive. Formally, the following implications should follow from the system assumptions Hyp:

$$Hyp \;\models\; Init \Rightarrow Inv \tag{2}$$

$$Hyp \;\models\; Inv \wedge BA_e(x) \Rightarrow Inv' \quad \text{for all events } e, \tag{3}$$

where *Init* denotes the predicate specifying the initial condition, *Inv* is the invariant predicate, and $BA_e(x)$ is the before-after predicate of event $e(x)$.

Second, the fairness condition associated with any event should imply that the event is actually feasible, for all states that satisfy the system invariant.

$$Hyp \models Inv \Rightarrow (fair_e(x) \Rightarrow \mathbf{fis}\, e(x)) \tag{4}$$

We have seen above that it can be reasonable to choose a fairness predicate that is strictly stronger than the feasibility predicate of an event. If $fair_e(x)$ could be true of some state where $e(x)$ is infeasible, it would be unreasonable to require $e(x)$ to occur eventually. Technically, such a specification would not be machine closed [Abadi and Lamport 1991].

It is easy to prove that these proof obligations are satisfied for the example system specification of Fig. 1.

Runs of event systems. A run of an event system is an ω-sequence

$$\sigma = s_0 \xrightarrow{e_0(d_0)} s_1 \xrightarrow{e_1(d_1)} s_2 \ldots$$

of states s_i (i.e., valuations of system variables) and event instances $e_i(d_i)$ where e_i is either an event declared for the event system or the special stuttering event τ such that the following conditions are satisfied:

- the initial condition holds of the initial state s_0,
- for each transition $s_i \xrightarrow{\tau} s_{i+1}$, the states s_i and s_{i+1} agree on the values of all system variables; such transitions are called stuttering steps,
- for each transition $s_i \xrightarrow{e_i(d_i)} s_{i+1}$ with $e_i(d_i)$ different from τ, the states s_i and s_{i+1} satisfy the before-after predicate $BA_{e_i}(d_i)$,
- σ satisfies all fairness conditions: for each event e and all parameters d, there are infinitely many positions $i \in \mathbb{N}$ such that either the fairness predicate $fair_e(d)$ is false at state s_i, or the transition $s_i \xrightarrow{e(d)} s_{i+1}$ occurs in σ.

For a well-formed specification, the proof obligations (2) and (3), and the requirement that the system invariant be a predicate formed from constants and system variables, ensure that the invariant is true at every state of a run.

2.2 Properties of Event Systems

We use a fragment of linear-time temporal logic to state and reason about properties of event systems. Specifically, we consider safety properties $\mathbf{stable}\, P$ and $\mathbf{inv}\, P$ where P is a state predicate, and liveness properties $F \leadsto G$ ("F leads to G") where F and G are first-order combinations of state predicates and event

$$\frac{P \wedge BA_e(x) \Rightarrow P' \quad \text{for all events } e(x)}{\mathbf{stable}\ P}\ (\text{stable})$$

$$\frac{Init \Rightarrow P \quad \mathbf{stable}\ P}{\mathbf{inv}\ P}\ (\text{induct}) \qquad \frac{\mathbf{inv}\ P \quad P \Rightarrow Q}{\mathbf{inv}\ Q}\ (\text{inv-weaken})$$

$$\frac{\begin{array}{c} P \Rightarrow fair_e(t) \\ P \wedge BA_a(x) \wedge \neg BA_e(t) \Rightarrow P' \vee Q' \quad \text{for all events } a(x) \end{array}}{P \leadsto Q \vee (P \wedge e(t))}\ (\text{fair})$$

$$\frac{\forall x \in S : F(x) \leadsto G \vee (\exists y \in S : y \prec x \wedge F(y)) \quad (S, \prec) \text{ well-founded}}{(\exists x \in S : F(x)) \leadsto G}\ (\text{wfo})$$

$$\frac{P \wedge BA_e(t) \Rightarrow Q'}{P \wedge e(t) \leadsto Q}\ (\text{effect}) \qquad \frac{\mathbf{inv}\ I \quad I \wedge F \leadsto G}{F \leadsto G \wedge I}\ (\text{inv-leadsto})$$

$$\frac{F \Rightarrow G}{F \leadsto G}\ (\text{refl}) \qquad \frac{F \leadsto G \quad G \leadsto H}{F \leadsto H}\ (\text{trans})$$

$$\frac{F \leadsto H \quad G \leadsto H}{F \vee G \leadsto H}\ (\text{disj}) \qquad \frac{F(x) \leadsto G(x)}{(\exists x : F(x)) \leadsto (\exists x : G(x))}\ (\text{exists})$$

Figure 2: Verification rules for fair event systems.

formulas $e(x)$. Given an ω-sequence $\sigma = s_0 \xrightarrow{e_0(d_0)} s_1 \xrightarrow{e_1(d_1)} s_2 \ldots$, these formulas are interpreted as follows:

$\sigma \models \mathbf{stable}\, P$ iff for all $n \in \mathbb{N}$, if $s_n \models P$ then $s_m \models P$ for all $m \geq n$
$\sigma \models \mathbf{inv}\, P$ iff $s_n \models P$ for all $n \in \mathbb{N}$
$\sigma \models F \leadsto G$ iff for all $n \in \mathbb{N}$, if $\sigma|_n \models F$ then $\sigma|_m \models G$ for some $m \geq n$

For the last clause, the notation $\sigma|_n \models F$ means that formula F holds at position n of the sequence σ: a state predicate P should hold at state s_n, an event formula $e(x)$ holds at position n if the transition $e_n(d)$ is caused by the event e and the parameters x evaluate to d in the current first-order interpretation. Elementary temporal formulas can be combined using standard propositional connectives and first-order quantifiers.

For an event-system specification Sys and a property φ, we write $Sys \models \varphi$ if $\sigma \models \varphi$ holds for every run σ of Sys, that is, if all runs of Sys satisfy φ.

Proof rules for deriving properties of fair event systems are shown in Fig. 2. They are similar to rules in formalisms such as Unity [Chandy and Misra 1988], TLA [Lamport 1994a] or linear-time temporal logic [Manna and Pnueli 1992]. In these rules, $Init$ denotes the initial condition of the system specification, and $BA_e(t)$ and $fair_e(t)$ denote the before-after predicate and the fairness assumption associated with the event $e(t)$. The variable x that appears in the rules (stable) and (fair) is assumed to be different from the free variables of P, Q, and $BA_e(t)$. Each rule should be read as asserting that the system satisfies the formula that appears as the consequent of the rule if it satisfies all the hypotheses. Assumptions on the constant parameters of a specification can be used in order to establish non-temporal hypotheses of these rules.

The rule (fair) is the basic rule for inferring liveness properties: it can be used to prove that any state satisfying a predicate P will be followed by a state satisfying Q, or a transition due to event $e(t)$. It has two hypotheses: one states that P should imply the fairness condition associated with event $e(t)$. The other hypothesis requires that whenever P holds, all possible successor states of all system transitions other than $e(t)$ must satisfy P or Q. In order to see that the rule is sound, assume that P is true at some point, but that Q never becomes true. Unless $e(t)$ occurs, the second hypothesis ensures that P remains true. By the first hypothesis, all these states also satisfy the fairness condition for $e(t)$. The fairness assumption adopted in the definition of runs (Sect. 2.1) implies that $e(t)$ must therefore eventually occur. Moreover, from the above we know that P must still be true at the (earliest) occurrence of $e(t)$, hence the consequent of the rule holds true. In practical applications, Q will often be **false**, and the rule is used to prove that the event $e(t)$ eventually occurs. It can be combined with rule (effect) in order to infer that a state satisfying some predicate P will eventually be followed by a state satisfying some target predicate. The rule (wfo) is useful for combining liveness properties by induction over a well-founded ordering $(S, \preceq)$. The rule (inv-leadsto) combines invariants and liveness properties. The remaining rules formalize Boolean and first-order reasoning with leads-to formulas.

3 Specifying Access Control

Event system specifications describe the functional behavior of systems. We now enrich system specifications by annotations that describe under which conditions events may or should occur. Typically, an access control policy specifies the actors (roles), objects (views), and operations (activities) of an information system, and then define which actors are allowed (or forbidden) to perform which operations on which objects. The OrBAC formalism [Abou et al. 2003] refines this idea by identifying the "organization" that enforces access control, and also by allowing the policy to refer to the "context" (an abstract view of the system state) in which an activity is performed. Roles, views, and activities can be organized in hierarchies, along which access rules are inherited [Cuppens et al. 2004].

The main advantage of formalisms such as OrBAC is that they provide a declarative language to specify access control, independently of a system implementation. The notion of context can be used to refer to the state of the system environment, such as the current time, but also to system-dependent information such as server load. It is conceptually obvious how an OrBAC model can be represented in an event system specification: the static structure of roles and views is translated into constant parameters, activities correspond to events, and context is represented by state predicates. We will not give a complete account of such a translation here, but will now propose an annotation of event system descriptions in order to describe access control policies. We consider the concepts of prohibitions, user rights, and obligations for specifying access control. Annotations of event system specifications with access control primitives can be used for different purposes. First, one may verify that the policy is implemented by a system specification or that it is preserved during system development. Second, one may wish to develop a monitor that enforces a policy over a fixed, underlying system. We focus on the first aspect in this paper.

3.1 Prohibitions

Fundamentally, access control policies specify constraints about when an activity is permitted, and when it is forbidden. It is conventional to state these constraints in the form of *permissions*: for example, if physicians are permitted to access a patient's files, all other roles are implicitly forbidden to do so. We will state such constraints in the form of *prohibitions* in order to make it more obvious that they restrict the occurrence of events, and in order to avoid confusion with the concept of user rights discussed in Sect. 3.2.

We propose to represent prohibitions by annotating event definitions with predicates. For the banking example, let us assume that there exists a risk function $risk(c, amt)$ that for a client c and an amount amt returns a risk assessment in $\{low, medium, high\}$. One might then specify

event $newLoan(c, l, amt, dur, mx)$
prohibition $risk(c, amt) = high$

to indicate that a new loan for a client must not be approved if the risk is considered high.

An event system implements the prohibition annotations if an event may occur only if it is not forbidden. Formally, we associate the following proof obligation with annotated event systems:

$$Hyp \models Inv \wedge \mathbf{fis}\, e(x) \Rightarrow \neg proh_e(x) \quad (5)$$

where $proh_e(x)$ is the prohibition predicate associated with event e, and, as before, Inv and Hyp denote the system invariant and the constant assumptions. If this condition can be proved of a system specification, the system guarantees that the event $e(x)$ never occurs in a run when it is forbidden by the security policy.

The event system of Fig. 1 does not implement the prohibition above because it does not evaluate the risk associated with a loan. Strengthening the definition of event *newLoan* by either of the formulas

$$risk(c, amt) = low \qquad \text{or} \qquad risk(c, amt) \in \{low, medium\}$$

will ensure that the system implements the prohibition expressed by the security policy. In general, the most permissive implementation is obtained by conjoining the predicate $\neg proh_e(x)$ to the before-after predicate of the event definition. Alternatively, the prohibition can be ensured by a run-time separate monitor that allows events to be activated only if the prohibitions are respected.

Observe, however, that strengthening the definition of an event by a predicate, such as $\neg proh_e(x)$, may invalidate the healthiness condition (4) that states that the fairness predicate of an event should imply its feasibility. We therefore add the following proof obligation to the well-formedness conditions of an event system with prohibitions:

$$Hyp \models Inv \wedge fair_e(x) \Rightarrow \neg proh_e(x). \tag{6}$$

This condition is trivially satisfied for the event *newLoan* of our running example, because no fairness is required of that event.

3.2 User rights

Prohibitions restrict when an event may occur. Dually, it may be interesting for a security policy to assert conditions under which the system must not disable an event. For example, we may wish to state explicitly that a client has the right to make extra payments within the agreed-upon limits:

event $extraPayment(l, amt)$
right $l \in loans \wedge amt \in \mathbb{Q} \wedge amt + extra(l) \leq maxExtra(l)$.

An event system implements a user right if the event is feasible whenever the predicate specifying the right holds:

$$Hyp \models Inv \wedge right_e(x) \Rightarrow \mathbf{fis}\, e(x) \tag{7}$$

Even if access control is implemented by a separate monitor, user rights should be verified over the underlying event system itself because the monitor only restricts the activation of events in system executions. Additionally, one should ensure that the monitor does not restrict an event from occurring when the user right holds true.

Obviously, prohibitions and user rights associated with an event should be mutually exclusive: a security policy is inconsistent if it grants a right to perform an activity when that activity is forbidden. Indeed, the conditions (5) and

(7) show that the user right and the prohibition predicate for an event e are never both true of any state that satisfies the system invariant. Prohibitions and rights need not be exhaustive: a security policy may choose to leave it unspecified in certain situations whether an activity may be performed or not. In particular, it is not unreasonable for a user right to be strictly stronger than the feasibility of the event. For example, a bank could accept payments beyond the pre-determined bound at its discretion.

User rights can be understood as specifying basic branching-time properties of a system: whenever the predicate $right_e(t)$ is true, the system has a possible continuation that begins with the event instance $e(t)$. This observation will become important in the context of refining event systems while preserving access control annotations, which we discuss in Sect. 4.2.

3.3 Obligations

Some languages for stating access control policies, such as OrBAC, also include primitives for specifying *obligations*. Intuitively, whereas a user right states when a certain activity *may* occur, an obligation asserts that the activity *should* occur. Because OrBAC does not consider a model of system execution, it does not formally specify how an obligation should be interpreted. Traditionally, concepts of permissions, rights, and obligations have been the domain of deontic logic [Hilpinen 1981, Meyer and Wieringa 1993]. To our knowledge, the interpretation of formulas of deontic logic over models of information systems such as event systems has not been studied, and we will continue to base our semantics on the familiar framework of event systems and their runs.

As for prohibitions and rights, we express obligations by predicates that annotate events. In our running example, we might want to assert that a user has an obligation to pay the rates for a loan, and therefore write

event $makePayment(l)$
obligation $l \in loans \wedge due(l) > 0.$

What does it formally mean for an event system to implement an obligation? A possible interpretation would be that whenever an obligation arises to perform event $e(t)$, that event should occur immediately, before the execution of any other event. However, this interpretation can easily lead to contradictions. For example, a user of a computer system may have an obligation to regularly change his password, but he can do so only when logged in. The obligation to change the password should not preclude the user from logging in, although it is conceivable that one could then prevent the user from doing anything but changing his password.

One could try to introduce a taxonomy of obligations, classifying them by their nature (legal, moral, technical, organizational etc.) or by their urgency. As a first approximation, we believe that obligations can usefully be understood as liveness properties, and can be formalized in temporal logic. The two following interpretations appear plausible:

$$\text{strict obligation: } \mathit{obl}_e(x) \leadsto e(x) \tag{8}$$

$$\text{weak obligation: } \mathit{obl}_e(x) \leadsto \neg\mathit{obl}_e(x) \lor e(x). \tag{9}$$

The strict interpretation of obligations requires that the event occurs eventually whenever the obligation arises. Under the weak interpretation, the obligation ceases as soon as the predicate $\mathit{obl}_e(x)$ becomes false, which needs not be due to an occurrence of the event $e(x)$. In our example, the weak interpretation appears more reasonable: for example, the obligation to pay a rate on a loan ceases when the loan is repaid via an extra payment. As an aside, we observe that the weak interpretation of an obligation coincides with the interpretation of a weak fairness requirement, with $\mathit{obl}_e(x)$ as the fairness predicate.

For either of the two interpretations above, the proof rules of Fig. 2 can be used to verify that a fair event system implements its obligations. The temporal interpretation of obligations may also be of interest when one is interested in deriving a security monitor that enforces obligations for a given system, at least for controllable events [Arnold et al. 2003], but we do not pursue this idea any further here.

In some applications, the interpretation of obligations as liveness properties may be too abstract, and it would be more natural to indicate real-time deadlines for obligations ("the payment should be received before the end of the current month"). We do not extend our method here to handle real-time requirements, but encoding real-time systems in event-based formalisms does not pose any conceptual problems [Abadi and Lamport 1994].

4 Refinement of System Specifications

In a refinement-based approach to system development, an initial high-level system specification is successively enriched by introducing implementation detail and new properties. The key requirement for a reasonable notion of refinement is that properties that have been established for an abstract model are preserved for refined models and do not have to be reproved. Formal verification accompanies the entire development process, and errors are discovered early, when it is relatively easy to correct them. Non-atomic refinements allow a developer to implement an event of a high-level model by a sequence of low-level transitions; they are the key to obtaining interesting differences in abstraction between the various models of a system, and must be supported by any practically useful method.

In this section, we define a refinement notion for fair event systems along standard lines, based on the preservation of traces. In this way, properties expressed in the temporal logic of Sect. 2.2 are preserved by refinement. We then study how refinement interacts with the annotations considered in Sect. 3 that represent access control primitives.

4.1 Refinement of fair event systems

Refinement notions for event systems that preserve safety properties are well known, and extensions for liveness and fairness properties have also been considered [Abrial and Mussat 1998, Back and Xu 1998, Barradas and Bert 2002, Darlot et al. 2003]. In the following, we make use of the language of temporal logic of Sect. 2.2 to state verification conditions that ensure that liveness properties are preserved. The use of temporal logic allows us to formulate our proof obligations at a higher level of abstraction than in traditional formulations.

Refined models describe the system at a finer level of granularity and typically introduce new events that have no observable effect at the previous levels of abstraction. Formally, we assume (without loss of generality) that the refinement is described with the help of a tuple var_{ref} of variables disjoint from the variables var_{abs} used in the original model. The two state spaces are related by a *gluing invariant* J, a state predicate built from the variables var_{abs} and var_{ref}, and the constant parameters of both models. We may assume that J implies both the abstract-level and the concrete-level invariants Inv_{abs} and Inv_{ref}. An event $ea(x)$ of the abstract model may be refined by any number of low-level events $er_1(x, y_1)$, $\ldots$, $er_n(x, y_n)$; for technical simplicity, we assume that all parameters of ea are also parameters of er_i. Additionally, new events $en(z)$ may be introduced in the refined model that do not have any counterpart in the abstract specification; in particular, non-atomic event refinements rely on such auxiliary low-level events.

An event system Ref is a refinement of an event system Abs with respect to the gluing invariant J if Ref is itself well-formed according to the conditions (2), (3), and (4), and if moreover all the following conditions hold (in the following, Hyp denotes the conjunction of the abstract- and concrete-level constant assumptions).

- For every initial state of the refinement there exists a corresponding initial state of the abstract specification:

$$Hyp \models Init_{ref} \Rightarrow \exists var_{abs} : Init_{abs} \wedge J \tag{10}$$

- Events of the refinement can be mapped to events or to stuttering transitions of the abstract specification. There are two cases:

- If event $er(x, y)$ refines an abstract event $ea(x)$ then its effect is similar to that of ea:

$$Hyp \;\models\; J \wedge BA_{er}(x, y) \Rightarrow \exists var'_{abs} : BA_{ea}(x) \wedge J' \tag{11}$$

- If event $en(z)$ is a new event then its effect is invisible at the abstract level[2]:

$$Hyp \;\models\; J \wedge BA_{en}(z) \Rightarrow \exists var'_{abs} : var'_{abs} = var_{abs} \wedge J' \tag{12}$$

– The refinement preserves the fairness constraints of the abstract level. Formally, assume that the abstract event $ea(x)$ is refined by low-level events $er_1(x, y_1), \ldots, er_n(x, y_n)$:

$$\begin{array}{rl} Ref \;\models & J \wedge fair_{ea}(x) \\ & \leadsto\; \vee\ (\exists y_1 : er_1(x, y_1)) \vee \ldots \vee (\exists y_n : er_n(x, y_n)) \\ & \quad\;\; \vee\ \neg\exists var_{abs} : J \wedge fair_{ea}(x) \end{array} \tag{13}$$

Condition (13) requires that any state in a run of the refinement that corresponds to a state satisfying the abstract fairness condition of event $ea(x)$ is followed either by the occurrence of one of the actions refining $ea(x)$, or by a state that no longer satisfies the abstract fairness condition. The proof obligation is conveniently expressed as a "leads-to" property that should be proved of the refined system, using the proof system of Fig. 2. This proof can make use of fairness conditions of Ref, as well as induction over well-founded orderings. In this way, a developer has more freedom in justifying a refinement than with the more traditional verification conditions of [Abrial and Mussat 1998, Back and Xu 1998] that are expressed in terms of variant functions.

Using a simulation argument that critically relies on the possibility of stuttering in the definition of runs of event systems, one obtains the following correctness theorem: every run of the refined event system Ref corresponds to a run of the abstract event system Abs, modulo the gluing invariant.

Theorem 1. *Assume that Ref is a refinement of Abs with respect to the gluing invariant J and that $\sigma = s_0 \xrightarrow{er_0} s_1 \xrightarrow{er_1} \ldots$ is a run of Ref. Then there is a run $\tau = t_0 \xrightarrow{ea_0} t_1 \xrightarrow{ea_1} \ldots$ of Abs such that J holds at the joint valuations obtained from s_i and t_i, for all $i \in \mathbb{N}$, and such that ea_i refines er_i whenever er_i refines an event of the abstract system, and is τ otherwise.*

[2] As suggested in [Abrial et al. 2005], this requirement could be weakened by requiring that event $en(z)$ merely preserves the high-level invariant.

Proof. We first construct an intermediate sequence $\rho = t_0 \xrightarrow{E_0} t_1 \xrightarrow{E_1} t_2 \ldots$ of states t_i and sets of instances of abstract events E_i; intuitively, any event instance in E_i can explain the transition from t_i to t_{i+1}. The sequence ρ and the set E_i are inductively constructed as follows:

– Because s_0 satisfies the initial condition $Init_{ref}$, condition (10) ensures that there exists some state t_0 that satisfies the initial condition $Init_{abs}$ and such that (s_0, t_0) jointly satisfy the gluing invariant J.

– For a transition $s_i \xrightarrow{er_i(d)} s_{i+1}$ of σ, we inductively assume that t_i has already been defined such that (s_i, t_i) jointly satisfy J. For the construction of E_i and t_{i+1}, there are two possible cases:

  - If the transition is due to the stuttering event τ or to an event instance $en(z)$ that does not have a counterpart in Abs, we let $t_{i+1} = t_i$. This choice ensures that (s_{i+1}, t_{i+1}) jointly satisfy J: in the case of a τ transition this follows because the states agree on all variables in $var_{abs} \cup var_{ref}$, which are the only free variables of J, in the case of a new event $en(d)$ it is ensured by condition (12). We let $E_i = \{\tau\}$.

  - Otherwise, the transition is due to an event instance $er(x, y)$ that refines some abstract event $ea(x)$, and t_{i+1} may be chosen according to the condition (11). In this case, we define

$$E_i = \{ea(d) : (t_i, t_{i+1}) \models BA_{ea}(d)\}$$

    as the set of instances of abstract events that could possibly explain the transition from t_i to t_{i+1}, this set is non-empty by condition (11).

Finally, the abstract run τ is obtained by choosing event instances $ea_i \in E_i$ such that all fairness conditions are satisfied: for any instance $ea(d)$ of an event ea of Abs such that there exists some $m \in \mathbb{N}$ such that $t_n \models fair_{ea}(d)$ for all $n \geq m$, the abstract event $ea(d)$ occurs infinitely often among the ea_i. This choice is possible since condition (13) ensures that concrete event instances $er(d')$ that refine $ea(d)$ occur infinitely often in σ, hence by condition (11) and the definition of E_i, the abstract event $ea(d)$ occurs infinitely often in E_i. Because overall there are only denumerably many instances of abstract events (using the assumption that the first-order universe is denumerably infinite), a standard combinatorial argument shows that $ea(d)$ can be chosen infinitely often. □

As a consequence, temporal logic properties can be transferred from an abstract event system Abs to its refinement Ref modulo the gluing invariant J. Formally, this is asserted by the following corollary.

Corollary 2. *Assume that Ref is a refinement of Abs with respect to the gluing invariant J and that $\sigma = s_0 s_1 \ldots$ is a run of Ref. If $\mathit{Abs} \models \varphi$ then $\mathit{Ref} \models \overline{\varphi}$ where $\overline{\varphi}$ is obtained from φ by replacing every positive occurrence of a non-temporal formula A by $\exists var_{abs} : J \wedge A$ and every negative occurrence by $\forall var_{abs} : J \Rightarrow A$.*

4.2 Refinement preserving access control

Let us now consider how refinement interacts with access control specifications. Assume that the event system Ref is a refinement of Abs with respect to the gluing invariant J. Also, assume that Abs was known to implement the prohibitions, obligations, and user rights associated with an abstract event $ea(x)$.

Prohibitions are represented by state predicates, and the condition (5) ensures that $\neg proh_{ea}(x)$ holds whenever event $ea(x)$ occurs in a run of Abs. Any concrete-level event $er(x, y)$ refining $ea(x)$ has to satisfy condition (11). Using the definition of feasibility (1) and first-order logic, it follows that $\overline{\neg proh_{ea}(x)}$ holds whenever event $er(x)$ occurs in a run of Ref. In other words, prohibitions are preserved modulo the gluing invariant J, and this is the preservation result that we may expect. The precise meaning of the translated property depends on the gluing invariant, but typical choices of J will imply that the abstract-level prohibitions are indeed preserved in the refined system.

Similarly, obligations have been interpreted as liveness properties, represented by the temporal logic formulas (8) or (9). Corollary 2 implies that a similar "leads to" formula is true of the refined model, again modulo translation along the gluing invariant. Therefore, obligations are preserved in the same sense as permissions and prohibitions.

These preservation results are not really surprising: we have interpreted prohibitions and obligations as safety or liveness properties of runs, and refinement of event systems is defined in such a way that properties of runs are preserved. However, we have also considered user rights, which were interpreted as branching properties in Sect. 3.2, and refinement of event systems does not necessarily preserve branching behavior.

For a concrete example, Fig. 3 proposes a refinement of the event *extraPayment* of our running example. Instead of an atomic event modeling an extra payment, the refinement introduces a protocol: the client has to apply for making an extra payment (event *demandPayment*), and this application can be approved or rejected by the bank, depending on the situation of the loan[3]. This refinement is acceptable according to the conditions (11) and (12) because *approvePayment* refines the abstract event *extraPayment* whereas the events *askPayment* and *rejectPayment* do not modify the abstract variables. However, the refinement does not imme-

[3] The new variable *extraDmd* is a multiset containing demands for extra payments; we use the symbols $\uplus$, $\Subset$, and $\Subset$ to denote multiset union, membership, and inclusion.

event $demandPayment(l, amt) =$
$\land\ l \in loans \land amt \in \mathbb{Q}$
$\land\ extraDmd' = extraDmd \uplus \{l \mapsto amt\}$
$\land$ UNCHANGED $loans, clt, due, rate, maxExtra, extra$
right $l \in loans \land amt \in \mathbb{Q}$

event $approvePayment(l, amt) =$
$\land\ (l \mapsto amt) \Subset extraDmd \land amt + extra(l) \leq maxExtra(l)$
$\land\ due' = due \oplus \{l \mapsto due(l) - amt\} \land extra' = extra \oplus \{l \mapsto extra(l) + amt\}$
$\land\ extraDmd' = extraDmd \setminus \{l \mapsto amt\}$
$\land$ UNCHANGED $loans, clt, rate, maxExtra$
fairness $(l \mapsto amt) \Subset extraDmd \land amt + extra(l) \leq maxExtra(l)$

event $rejectPayment(l, amt) =$
$\land\ (l \mapsto amt) \Subset extraDmd \land amt + extra(l) > maxExtra(l)$
$\land\ extraDmd' = extraDmd \setminus \{l \mapsto amt\}$
$\land$ UNCHANGED $due, extra, loans, clt, rate, maxExtra$
fairness $(l \mapsto amt) \Subset extraDmd \land amt + extra(l) > maxExtra(l)$

Figure 3: Refining event *extraPayment*.

diately preserve the user right

event $extraPayment(l, amt)$
right $l \in loans \land amt \in \mathbb{Q} \land amt + extra(l) \leq maxExtra(l).$

considered in Sect. 3.2: the precondition $(l \mapsto amt) \Subset askExtra$ of the concrete-level event is not implied by the predicate describing the user right. Preservation of user rights therefore requires some additional consideration.

A first idea would be to impose the condition

$$Hyp \models Inv_{ref} \land \overline{right_{ea}(x)} \Rightarrow (\exists y_1 : \textbf{fis}\, er_1(x, y_1)) \lor \ldots \lor (\exists y_n : \textbf{fis}\, er_n(x, y_n)) \quad (14)$$

where again $er_1(x, y_1)$, ..., $er_n(x, y_n)$ are the concrete-level events corresponding to the abstract event ea. This condition requires that whenever the abstract-level right holds in a concrete run (modulo the gluing invariant), some event refining the abstract event is executable. Obviously, this preserves user rights, but it would rule out the refinement of Fig. 3. More generally, this condition appears too strong in the context of a refinement notion that changes the granularity of atomicity. Recall that a single abstract event ea can be implemented in the refinement by a sequence of concrete events, all but the last of which are invisible at the abstract level. As in the example of Fig. 3, a concrete event er refining the abstract event ea is typically not immediately feasible whenever ea is, because it needs to be prepared by auxiliary events that are unobservable at the abstract level. We therefore believe that a more useful condition for refining user rights is to require a combination of concrete-level user rights that ensure that the branch leading to er can be started and concrete-level obligations that ensure that er will then eventually occur.

Formally, assume that the abstract system specification contains an event $ea(x)$ for which we wish to ensure a user right $right_{ea}(x)$. Also assume that $ea(x)$ is refined by the concrete-level events $er_1(x,y_1), \ldots, er_n(x,y_n)$. We then require that the implementation Ref contains events $ei_1(x,z_1), \ldots, ei_m(x,z_m)$ with user rights specified by $right_{ei_j}(x,z_j)$ such that

$$\overline{right_{ea}(x)} \Rightarrow (\exists z_1 : right_{ei_1}(x,z_1)) \vee \ldots \vee (\exists z_m : right_{ei_m}(x,z_m)) \quad \text{and} \tag{15}$$

$$ei_j(x,z_j) \leadsto \neg\overline{right_{ea}(x)} \vee (\exists y_1 : er_1(x,y_1)) \vee \ldots \vee (\exists y_n : er_n(x,y_n)) \tag{16}$$

Condition (16) applies for all $j = 1, \ldots, m$; the disjunct $\neg\overline{right_{ea}(x)}$ on the right-hand side of (16) corresponds to a weak interpretation of obligations.

The above conditions, together with the interpretations of the user rights for the refined specification, imply that whenever the translated abstract user right holds at some point during a concrete-level run, the refinement ensures the right to start a branch that will eventually lead to the occurrence of an event er refining the original event ea, provided the abstract-level right persists. The obligation to perform the event er ceases when the abstract right $\overline{right_{ea}(x)}$ does not persist—for example, this can arise due to the concurrent exercise of another right.

Back to the example of Fig. 3, we claim that this refinement respects the abstract-level user right because it satisfies the conditions (15) and (16). We assume that the gluing invariant contains the conjunct

$$extraDmd \subseteq loans \times \mathbb{Q}$$

that asserts the "type correctness" of the new variable $extraDmd$. We choose $demandPayment$ for the auxiliary event ei, and condition (15) boils down to proving

$$l \in loans \wedge amt \in \mathbb{Q} \wedge amt + extra(l) \leq maxExtra(l) \Rightarrow l \in loans \wedge amt \in \mathbb{Q}$$

which is trivial. On the other hand, condition (16) requires us to show

$$\begin{aligned} & demandPayment(l,amt) \leadsto \\ & \quad \vee \neg(l \in loans \wedge amt \in \mathbb{Q} \wedge amt + extra(l) \leq maxExtra(l)) \\ & \quad \vee approvePayment(l,amt) \end{aligned}$$

and this property is ensured by the fairness condition for event $approvePayment$. Whereas the abstract user right is preserved, the client cannot cheat on the bank by demanding two extra payments that together would exceed the allowed limit:

although a client may always *ask* for an extra payment (including in the time between applying for a payment and the approval or rejection by the bank), the bank's obligation to approve extra payments ceases when the limit has been reached, so it is free to reject a second application for extra payments. This appears as a reasonable refinement of the abstract-level user right of Sect. 3.2.

5 Conclusion

Event systems are a convenient and widely accepted framework for modeling information systems. In particular, properties of their runs can be derived using well-known rules, and refinement concepts for event systems are well established. In this paper, we have considered an extension of event systems by considering access control primitives and have proposed to annotate events with predicates to specify prohibitions, obligations, and user rights.

We have taken our inspiration from existing, declarative languages for describing access control, such as OrBAC, that identify the static structure of an information system, including the roles, the views, and the activities, and then spell out the conditions under which activities may, must, or must not be performed. We go beyond OrBAC by interpreting these conditions over the specification of system behavior, and have identified proof rules for verifying that a system implements a security policy. It appears obvious to interpret prohibitions as safety properties that constrain the occurrences of events. We also propose that obligations be interpreted as liveness properties, at least at an abstract level of specifications, and we have used a simple temporal logic to state and prove such properties of event systems. As a final category of access control primitives, we have considered user rights, which can be interpreted as elementary branching properties of systems.

Development methods based on stepwise refinement have traditionally been associated with event systems. They allow a designer to develop a system as the outcome of a sequence of models that progressively introduce details in the system representation, and add correctness properties. The cornerstone of refinement is the preservation of properties that have been established for abstract models. Standard refinement concepts preserve traces of models, and this ensures that prohibitions and obligations are preserved across refinement. Branching properties, including user rights, are not guaranteed to be preserved, and we have proposed additional conditions that rely on a combination of concrete-level rights and obligations. Our condition is compositional in the sense that a model can be further refined by implementing the lower-level rights in the same manner; the obligations are guaranteed to be preserved by the ordinary notion of system refinement.

More experience will be necessary to evaluate whether our notions are useful and feasibility in practice. An interesting and complementary approach to verifying access control properties would be to synthesize security monitors that enforce a security policy over a given underlying information system. The study of refinement notions for more advanced concepts in information security, including information flow and confidentiality, is a challenging topic for future work that has already attracted some attention [Santen 2006, Hutter 2006].

References

[Abadi and Lamport 1991] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.

[Abadi and Lamport 1994] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, Sept. 1994.

[Abou et al. 2003] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization based access control. In *4th Intl. IEEE Workshop Policies for Distributed Systems and Networks*, pages 120–131, Como, Italy, 2003. IEEE Press.

[Abrial 1996] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abrial 2003] J.-R. Abrial. $B^{\#}$: Toward a synthesis between Z and B. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *Formal Specification and Development in Z and B (ZB 2003)*, volume 2651 of *Lecture Notes in Computer Science*, pages 168–177. Springer Verlag, 2003.

[Abrial et al. 2005] J.-R. Abrial, D. Cansell, and D. Méry. Refinement and Reachability in Event_B. In H. Treharne, S. King, and M. Henson, editors, *Formal Specification and Development in Z and B (ZB 2005)*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241, Guilford, UK, 2005. Springer Verlag.

[Abrial and Mussat 1998] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Advances in the Development and the Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer Verlag, 1998.

[Arnold et al. 2003] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controlers with partial observation. *Theoretical Computer Science*, 303(1):7–34, 2003.

[Back and von Wright 1998] R. Back and J. von Wright. *Refinement calculus—A systematic introduction.* Springer Verlag, 1998.

[Back and Xu 1998] R.-J. Back and Q. Xu. Refinement of fair action systems. *Acta Informatica*, 35:131–165, 1998.

[Barradas and Bert 2002] H. R. Barradas and D. Bert. Specification and proof of liveness properties under fairness assumptions in B event systems. In M. J. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 360–379, Turku, Finland, 2002. Springer Verlag.

[Benferhat et al. 2003] S. Benferhat, R. E. Baida, and F. Cuppens. A stratification-based approach for handling conflicts in access control. In *8th ACM Symp. Access Control Models and Technologies (SACMAT'03)*, 2003.

[Bertino et al. 2003] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[Chandy and Misra 1988] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Reading, Mass., 1988.

[Cuppens et al. 2005] F. Cuppens, N. Cuppens-Boulahia, and J. Garcia. Misconfiguration management of network security components. In *IASTED International Conference on Communication, Network, and Information Security (CNIS 2005)*, 2005.

[Cuppens et al. 2004] F. Cuppens, N. Cuppens-Boulahia, and A. Miège. Inheritance hierarchies in the Or-BAC model and application in a network environment. In A. Sabelfeld, editor, *Proc. Foundations of Computer Security (FCS04)*, pages 41–60, Turku,Finland, 2004. Turku Center for Computer Science, Report G-31.

[Darlot et al. 2003] C. Darlot, J. Julliand, and O. Kouchnarenko. Refinement preserves PLTL properties. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 408–420, Turku, Finland, 2003. Springer Verlag.

[Guelev et al. 2004] D. P. Guelev, M. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Proc. 7th Intl. Conf. Information Security (ISC 2004)*, volume 3225 of *Lecture Notes in Computer Science*, pages 219–230, Palo Alto, CA, 2004. Springer.

[Hilpinen 1981] R. Hilpinen, editor. *New Studies in Deontic Logic: Norms, Actions, and the Foundations of Ethics*, volume 152 of *Synthese Library*. D. Reidel, Dordrecht, Holland, 1981.

[Hutter 2006] D. Hutter. Possibilistic information flow control in MAKS and action refinement. In G. Müller, editor, *Intl. Conf. Emerging Trends in Information and Communication Security (ETRICS 2006)*, volume 3995 of *Lecture Notes in Computer Science*, pages 268–281, Freiburg, Germany, 2006. Springer Verlag.

[Jürjens 2004] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[Koch et al. 2005] M. Koch, L. V. Mancini, and F. Parisi-Presicce. Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, 71(1):1–33, 2005.

[Lamport 1994a] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lamport 1994b] L. Lamport. How to write a long formula. *Formal Aspects of Computing*, 6(5):580–584, 1994.

[Manna and Pnueli 1992] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems—Specification*. Springer Verlag, New York, 1992.

[McLean 1992] J. D. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–57, 1992.

[Meyer and Wieringa 1993] J.-J. Meyer and R. Wieringa, editors. *Deontic Logic in Computer Science*. Wiley, 1993.

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

[Santen 2006] T. Santen. A formal framework for confidentiality-preserving refinement. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *Europ. Symp. Research in Computer Security (ESORICS 2006)*, volume 4189 of *Lecture Notes in Computer Science*, pages 225–242, Hamburg, Germany, 2006. Springer Verlag.

[Zhang et al. 2005] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In J. Zhou, J. Lopez, R. H. Deng, and F. Bao, editors, *Proc. 8th Intl. Conf. Information Security (ISC 2005)*, volume 3650 of *Lecture Notes in Computer Science*, pages 446–460, Singapore, 2005. Springer.