Verification and Enforcement of Access Control Policies

Antonio Cau · Helge Janicke · Ben Moszkowski

Received: date / Accepted: date

Abstract Access control mechanisms protect critical resources of systems from unauthorized access. In a policy-based management approach, administrators define user privileges as rules that determine the conditions and the extent of users' access rights. As rules become more complex, analytical skills are required to identify conflicts and interactions within the rules that comprise a system policy – especially when rules are stateful and depend on event histories. Without adequate tool support such an analysis is error-prone and expensive. In consequence, many policy specifications are inconsistent or conflicting that render the system insecure. The security of the system, however, does not only depend on the correct specification of the security policy, but in a large part also on the correct interpretation of those rules by the system's enforcement mechanism.

In this paper, we show how policy rules can be formalized in Fusion Logic, a temporal logic for the specification of behavior of systems. A symbolic decision procedure for Fusion Logic based on Binary Decision Diagrams (BDDs) is provided and we introduce a novel technique for the construction of enforcement mechanisms of access control policy rules that uses a BDD encoded enforcement automaton based on input traces which reflect state changes in the system. We provide examples of verification of policy rules, such as absence of conflicts, and dynamic separation of duty and of the enforcement of policies using our prototype implementation (FLCheck) for which we detail the underlying theory.

Keywords Access control policy · Policy Enforcement · Policy Verification · Binary Decision Diagram

A. Cau · H. Janicke · B. Moszkowski
Software Technology Research Laboratory
De Montfort University
LE1 9BH Leicester, UK
Tel.: ++44 (0)116 257 7937
Fax: ++44 (0)116 257 7936
E-mail: acau@dmu.ac.uk, heljanic@dmu.ac.uk, benm@dmu.ac.uk

1 Introduction

The management of security in large systems is a complex task that requires system administrators to specify constraints on the usage of system resources. Policy-based management [4, 63] is a modular approach that keeps the specification and enforcement of these constraints loosely coupled from the system. Constraints are expressed in form of a *policy*, typically a collection of event-condition-actions rules that determine actions to be taken by enforcement mechanisms. A commonly used architecture, e.g., in [18, 27, 53], for policy-based management is defined in [29], separating the decision making (Policy Decision Point) and the concrete enforcement (Policy Enforcement Point) of that decision on the system. Here the *policy* determines the behavior of the Policy Decision Point (PDP), e.g., whether an access request is permitted or denied. The specification and verification of the policy is therefore crucial for the administration of the system.

In previous work [31, 33, 62], we presented a formal policy specification language for history-based policies and have shown how these policies can be refined towards executable enforcement code [32]. History-based policies [1] are a very expressive class of policies that can define policy decisions dependent on previously observed behaviors within the system. Whilst this has immediate advantages, e.g., for the expression of dynamic separation of duty constraints, it also adds to the complexity of specification and verification of such policies, especially if such policies are to be enforced in a truly concurrent setting.

A key requirement to successful administration using the policy-based management approach is that conflicts in policies can be detected before the policy is pushed out to the PDPs in the system. Whilst traditional conflict analysis of policies [44] is in theory well understood, history-based policies offer some additional challenges that relate to the inter-dependencies between past observations and policy decision making. These may require additional synchronization constraints between several concurrent Policy Decision Points deployed in the system.

The contribution of this paper is twofold:

- We show how a history-based policy can be specified in Fusion Logic, allowing us to verify properties of the policy specification, e.g., accessibility, dynamic separation of duty, dynamic and static conflicts.
- We describe a decision procedure for Fusion Logic we have developed that uses Binary Decision Diagrams (BDDs), and show how this decision procedure can be adapted to act as an enforcement mechanism for history-based access control policies.

The advantage of our approach is that our formalism can be used for both the analysis and the enforcement of policies yielding a verifiable framework for policy enforcement that relies on logic as opposed to reference implementations of software [27, 53]. Figure 1 provides an overview of our specification/verification and enforcement framework.

The framework consists of three components: Specification, Verification and Enforcement. In the Specification component, one specifies a system by a set of history-based access control rules. In the Verification component, one takes this set of rules together with properties that need to hold and check whether this set of rules satisfy these properties. In the Enforcement component, the set of rules plus inputs, acting as triggers for these rules, are evaluated to determine whether to grant or deny a particular access.

Woo and Lam [66] 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. [30] have addressed this problem and

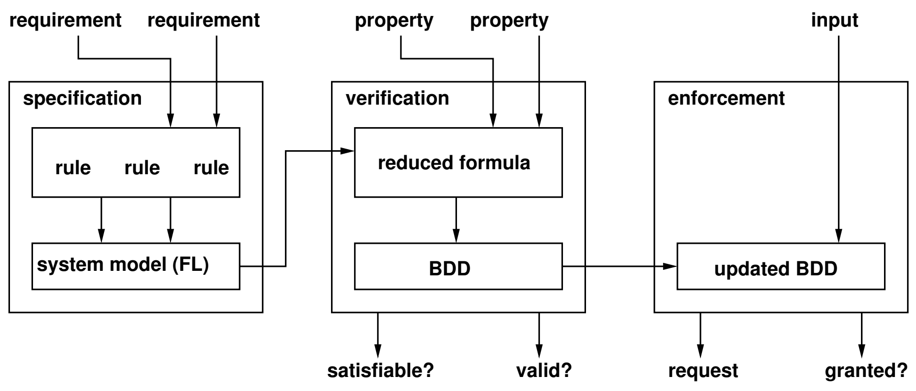


Fig. 1 Overview

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 [5], permissions are often limited in time or may hold for specific periods of time. An interval-based temporal model for access control has been proposed by Bertino et al. [5]. 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 the framework cannot handle the enforcement of multiple policies.

Usage Control (UCON) Models [59, 69], similar to Access Control Models, control and govern the users' access to resources and services that are available in the system. One of the major improvements of UCON over traditional access control models is the continuity of the control and the concept of attribute mutability. The formalization, however, makes a strong assumption in that only a single usage process is specified. It is assumed that the time-line is finite, i.e., it starts with the beginning of the single usage request and ends before the subsequent usage request. This makes it difficult to reason about the interactions of several *concurrent* usage requests, or even *sequences* of usage requests and so complicates the formal analysis of policies.

The remainder of this paper is organized as follows: In Section 2 we discuss history-based access control policies and their underlying computational model. In Section 3 we cover the specification of policies and the formalization of the computational model. We present Fusion Logic and discuss its syntax and semantics. We show how access control requirements can be captured as rules in Fusion Logic and how a system model can be obtained. The system model is the starting point for verification and enforcement. In Section 4 we present an automatic way of checking the satisfiability and validity of Fusion Logic formulae. We also describe how one can derive enforcement mechanisms for history-based access control policies, expressed in Fusion Logic, based on the same techniques that are used to determine the satisfiability/validity of Fusion Logic formulae. In Section 5 we illustrate the verification of properties and the derivation of an enforcement mechanism for a policy-based system with the help of case studies. We conclude our paper in Section 6.

2 Access control policies

We will introduce the computational model that will be used to characterize systems that are governed by history-based access control rules. Furthermore, we compare our model with the Flexible Authorization Manager authorization language (FAM) of Jajodia et al. [30], the Usage Control (UCON) policy language of Sandhu et al. [55, 69], and the Temporal Role-Based Access Control (TRBAC) model of Bertino et al. [5, 34, 35, 46].

2.1 Computational model for policy based management

Our computational model presented here describes the entities that comprise the system, their behaviors, and interactions. It represents a suitable abstraction for many real-world implementations that use the Policy Decision Point (PDP) Policy Enforcement Point (PEP) architecture [29, 53] to implement policy-based management. For the purpose of specification, verification, and analysis of dynamic security policies, the externally observable behavior of a system, i.e., the sequence of actions it does perform, is sufficient. We therefore refrained from modeling implementation details of the domain-dependent interactions between users and system.

We distinguish three different entities in the system: *subjects*, *objects*, and *reference monitors*. *Subjects* are entities, that (pro-)actively perform actions that affect objects. A subject can be a human user, role, or a program acting on behalf of a user. *Objects* are passive entities that represent shared data-structures in the information system. *Reference monitors* control the access to objects and determine whether a specific action can be performed by a subject or not. The concrete conditions under which a reference monitor *permits* an execution request or *denies* it are specified in the security policy.

The security policy represents an abstract specification of constraints on the interactions between the subjects and objects in the system. The abstract specification is then constructively refined into the behavior of the reference monitor such that the overall system satisfies the policy. Proving properties of the policy means that ensuring these properties are preserved by the system if the implementation of the reference monitor is *correct*, i.e., they adhere to the constraints specified in the policy.

The behavior of a reference monitor and its interaction with the other system components are detailed in Figure 2 as a Statechart [23]. Statecharts constitute an extensive generalization of state-transition diagrams. They allow for multilevel states decomposed in an And/Or fashion, and thus support economical specification of concurrency and encapsulation. Concurrency is represented by a dashed line that separates components of a parallel system. The labels on the transitions in Statecharts are of the form $Trigger[Condition]/Action$, where *Trigger* determines if and when a transition will be taken and *Action* is performed when a transition is taken and the *Condition* is true. An action includes the generation of events.

User Model: Every subject S represents a user process (see User process in Figure 2) acting on behalf of a user. This process can be either in a state *idle*, *wait*, or *access*. Initially, we assume a user process S to be in its *idle* state. By raising the event *Req(s,o,a)*, the process S indicates that it requests the execution of action a on the system object o and moves to the state *wait*. It will remain in the waiting state until it is either denied (event *Deny(s,o,a)*) or the request is executed (event *Exec(s,o,a)*).

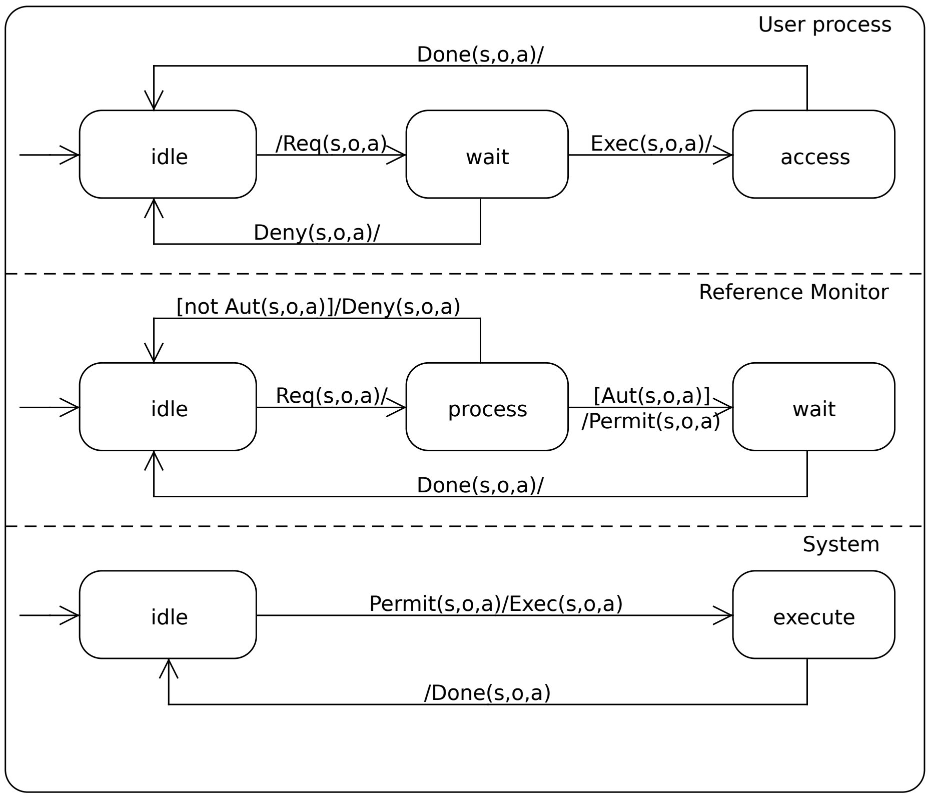


Fig. 2 Computational Model

Reference Monitor Model: The reference monitor (RM), as depicted in Figure 2, is a process that is initially in its *idle* state. Upon a user request *Req(s,o,a)*, the RM moves into the state *process* in which its behavior is specified by the policy. If the policy grants access (Aut(s,o,a) is true) it will raise the event *Permit(s,o,a)*; if it denies the access (Aut(s,o,a) is false), the event *Deny(s,o,a)* is raised. The RM subsequently returns to its *idle* state.

System Model: The access to the objects is facilitated by the system process, depicted in Figure 2. We assume that the system is initially in the state *idle*. On the event that the controller permits the execution, it will move to the state *execute* and raise the event *Exec(s,o,a)* that synchronizes the states *access* of the user process and the state *execute* of the system. The concrete behaviors of the user process and the system in these states are not explicitly defined.

The computational model represents a simplification of real information systems, where not only subjects can concurrently make requests, but also the reference monitors and the system facilitating access to the shared objects are distributed and can exhibit concurrent behavior.

2.2 History-based policies

Policies constrain the behavior of the reference monitors in the information system. More precisely, *access control* policies determine the choice of the reference monitor to permit or deny the execution of a request. A complete specification of the reference monitor can be given in the form of an access control matrix [42] that fully determines the access rights at any point in time during the system execution. Example 1 gives a simple access control matrix.

Example 1 Let $\mathrm{Aut}(s,o,a)$ denote that subject s is allowed to perform action a on object o. Assume we have subjects $\mathbf{nurse}_0$ and $\mathbf{nurse}_1$, actions **read** and **write**, and object **epr** (electronic patient record). The following matrix

$\mathrm{Aut}(s,\mathbf{epr},a)$	**read**	**write**
$\mathbf{nurse}_0$	true	false
$\mathbf{nurse}_1$	false	false

describes that $\mathbf{nurse}_0$ is allowed to **read** the **epr**, but $\mathbf{nurse}_1$ is not allowed to **read** it. Furthermore, both $\mathbf{nurse}_0$ and $\mathbf{nurse}_1$ are not allowed to **write** to the **epr**.

As we are interested in history-based access control [1], this matrix will depend not only on the current state of the information system, but also on the history of execution. We use a rule-based approach and specify (sets of) access control rights in terms of policy rules.

Our framework uses a rule-based conflict resolution approach similar to the one presented by [30] that allows for the expression of hybrid policies, i.e., policies that can express both positive and negative authorizations. Conflicts in the specification are eliminated by dedicated decision rules that define the precedence of positive and negative authorizations with respect to each other. Policy rules define the behavior of the following access control variables:

$\mathrm{Aut}^{+}(s,o,a)$:	Subject s is authorized to perform action a on object o.
$\mathrm{Aut}^{-}(s,o,a)$:	Subject s is not authorized to perform action a on object o.
$\mathrm{Aut}^{d}(s,o,a)$:	Subject s is authorized to perform action a on object o in case of conflicts.

The superscript $+$ here indicates a *positive* authorization, $-$ indicates a negative authorization, and d indicates a decision rule[1]. By employing decision rules for every access control request, the policy decision is conclusive. Ideally, however, the policy writer should be aware of potentially conflicting rules in order to adequately define these decision rules.

In our work, the operator *always-followed-by* [31, 33, 62], denoted by $Pre \mapsto W$, will be used to describe history-based access control policies. This operator captures the relation between the premise of a rule Pre and its consequence W. The intuition is that whenever Pre holds for a history of states, then W holds in the last state of that history. The consequence W can only contain access control variables, while the premise Pre is a formula capturing the history leading to the consequence. Example 2 describes informally various sample policies.

Example 2 Let us illustrate simple policies by an example. Consider a health care service where mr stands for a medical record, $owner(pat,mr)$ is a predicate denoting that patient pat is the owner of medical record mr, the predicate $treated(pat,phy)$ means that the patient pat is being treated by the physician phy, and $role(S,nurse)$ means that subject S is a

[1] For readability we use Aut^{d} in the context of Policy, and Aut in the context of the Enforcement Mechanism.

nurse. Here are some policy rules with their informal description. pat, mr, and phy are free variables ranging over respectively, the set of patients, medical records, and physicians. The free variable S ranges over the set of all subjects. The semantics of policy rules is such that one universally quantifies over the free variables appearing in a policy rule. The constants in a policy rule are indicated by a bold font.

– Patients are allowed read access to their medical records.
  $owner(pat,mr) \mapsto \mathrm{Aut}^{+}(pat,mr,\mathbf{read})$
– Physicians are allowed to read their patients' medical records.
  $treated(pat,phy)$ and $owner(pat,mr) \mapsto \mathrm{Aut}^{+}(phy,mr,\mathbf{read})$
– Nurses are forbidden read access to patient records.
  $role(S,\mathbf{nurse}) \mapsto \mathrm{Aut}^{-}(S,mr,\mathbf{read})$

Conflicting authorizations can be derived, for instance, when a nurse is at the same time a patient in the clinic. In this case, the first rule states that (s)he is allowed to read his/her record, while the third rule forbids him/her to do so. The conflict resolution mechanism is specified by the following policy rule that gives precedence to permission in the event of such a conflict. So, the policy does allow nurses read access to their own personal health information.

– Allowed access takes precedence over denied access.
  $\mathrm{Aut}^{+}(S,mr,\mathbf{read}) \mapsto \mathrm{Aut}^{d}(S,mr,\mathbf{read})$

The following example we list policy rules we would like to express in our policy specification language.

Example 3 We wish to express access control decisions based on past observed behavior of the system.

– Conditional on the current state:
  $0 : (owner(s,o)$ and $account(o)) \mapsto \mathrm{Aut}^{+}(s,o,\mathbf{withdraw})$
  where $n : W$ denotes that W holds n states from the end of the history. Only the current owner of a bank account can withdraw money.
– Conditional on the history:
  sometimes $done(s,o,a) \mapsto \mathrm{Aut}^{-}(s,o,a)$
  A subject must not perform an action on the same object twice. After the action has been executed once, all further requests will be denied. The condition is evaluated over the whole execution history. sometimes here should check whether in any suffix of the history the requested action has been done. $done(s,o,a)$ refers to the event raised by the system when an authorized action has been successfully performed. The use of the same variable names in the consequence and the premise of the rule means that the same subject, object, and action are referred to.
– Conditional on the history:
  sometimes $done(s_2,o,a)$ and $s_1 \neq s_2 \mapsto \mathrm{Aut}^{-}(s_1,o,a)$
  A subject is denied to perform an action if this action has already been performed by another subject. The above rule allows to model exclusive resource access.
– Conditional on the history:
  sometimes $(\mathrm{signin}_{\mathrm{hj}}\,;$ always(not $\mathrm{signout}_{\mathrm{hj}})) \mapsto \mathrm{Aut}^{+}(\mathbf{hj},\mathbf{door},\mathbf{open})$
  where always W denotes that for every state W holds and ';' denotes the sequential composition of two histories. Allow the user *hj* to open doors as long as he has registered his visit at the reception desk.

– Invariant:
  $\mathsf{always\ not}\ \mathit{bankrupt}(s) \mapsto \mathrm{Aut}^+(s, \mathbf{loan}, \mathbf{take})$
  A subject is only allowed to take a loan if (s)he was never bankrupt.
– Choice:

$$\left(\begin{array}{l} s_1 \neq s_2\ \mathsf{and}\ \mathit{parent}(s_1,s)\ \mathsf{and}\ \mathit{parent}(s_2,s)\ \mathsf{and} \\ ((\mathit{age\_less\_10}(s)\ \mathsf{and\ sometimes}\ \mathit{done}(s_1,o,\mathbf{consent}(a))\ \mathsf{and} \\ \mathsf{sometimes}\ \mathit{done}(s_2,o,\mathbf{consent}(a)))\ \mathsf{or} \\ (\mathsf{not}\ \mathit{age\_less\_10}(s)\ \mathsf{and}\ (\mathsf{sometimes}\ \mathit{done}(s_1,o,\mathbf{consent}(a))\ \mathsf{or} \\ \mathsf{sometimes}\ \mathit{done}(s_2,o,\mathbf{consent}(a))))) \end{array}\right) \mapsto \mathrm{Aut}^+(s,o,a)$$

  For a child younger than 10, both parents need to give consent, otherwise one parent's consent suffices.
– Collaboration:

$$\left(\begin{array}{l} 5 : (\mathsf{sometimes}\ \mathit{req}(s_1,\mathbf{door},\mathbf{open})\ \mathsf{and} \\ \quad \mathsf{sometimes}\ \mathit{req}(s_2,\mathbf{door},\mathbf{open})\ \mathsf{and}\ s_1 \neq s_2) \end{array}\right) \mapsto \mathrm{Aut}^+(s,\mathbf{door},\mathbf{open})$$

  The door can only be opened if at least two subjects requested the door to be opened within the last 5 states of the history. Notice that in the above rule the outcome of the previous request is not decisive, i.e., even if both previous requests were denied, the condition is met. The order in which these requests were made is arbitrary and the requests could even be made concurrently.
– Sequential access:

$$\left(\begin{array}{l} \mathsf{not}(\mathsf{sometimes}\ \mathit{done}(s,\mathbf{invoice},\mathbf{receive}); \\ \quad \mathsf{sometimes}\ \mathit{done}(s_a,\mathbf{invoice},\mathbf{authorize}))) \\ \mathsf{and}\ 0 : \mathit{role}(s_a,\mathbf{accountant}) \end{array}\right) \mapsto \mathrm{Aut}^-(s,\mathbf{bank},\mathbf{pay}(\mathbf{invoice}))$$

  An invoice cannot be payed unless it has been received and was authorized by an accountant. This rule enforces a sequence of two actions (**receive** and **authorize**) to be successfully performed prior to the **invoice** being payed.
– Counting states in the history:
  $\mathsf{exists}\ 0 <= i <= 3\ (i : \mathit{deleg}(s,s',o,a))\ \mathsf{and}\ 0 : \mathrm{Aut}^+(s,o,a) \mapsto \mathrm{Aut}^+(s',o,a)$
  If a right has been delegated in the last 4 states of the history, and the person delegating currently holds this right, the same right is also held by the delegatee. In this example delegation is captured by the event $\mathit{deleg}(s,s',o,a)$ that denotes the delegation of the right a on o from s to s'. The defined period of time is here assumed to be 4 states.
– Cardinality on history:
  $\mathsf{sometimes}\ ((\mathit{req}(s,o,a)\,;\mathsf{sometimes}\ \mathit{req}(s,o,a))^*) \mapsto \mathrm{Aut}^+(s,o,a)$
  where f^* denotes the sequential composition of f a finite number of times. A subject s is allowed to perform action a on object o if an even number of requests have been made before.

2.3 Comparison with other access control models

Our policy language has similarities with the Flexible Authorization Manager authorization language (FAM) of Jajodia et al. [30] in that both permissions and denials can be explicitly stated and conflict resolution takes place using dedicated decisions rules. The key difference is that the premise of rules is not only a condition on the current state of the system, but it can describe a set of behaviors that when observed will trigger the rule. In terms of expressiveness, the policy language is similar to Usage Control (UCON) models of [55, 69]. In [5], a Temporal Role-Based Access Control (TRBAC) model is introduced that enables one to express temporal constraints for an Role-Based Access Control model. With these constraints one can restrict users to assume roles only at predefined time periods. In the following, we will discuss the UCON, FAM, and TRBAC access control models in more detail.

2.3.1 Flexible Authorization Manager authorization language

The work by Jajodia et al. [30] identifies that most policy languages allow only the specification of two specific types of policies. These are namely:

– Open Policies: Everybody is allowed access unless denied. A blacklist is a typical example.
– Closed Policies: Everybody is denied access unless explicitly allowed. This is for example the case for Java policy files.

Jajodia et al. then propose the use of positive and negative authorizations rules and show how conflicts are resolved by decision rules. They introduce the rule-based Flexible Authorization Manager (FAM) authorization language. Here positive and negative authorizations for a subject (or group) to perform an action on a specific object can be expressed. The example below shows a positive and a negative authorization rule:

$$\begin{array}{l} cando(file, s, +read) \leftarrow in(s, Employee) \\ cando(file, s, -read) \leftarrow in(s, Employee). \end{array}$$

This rule defines that if subject *s* is a member of the group *Employee*, then it can read the file. Negative authorization would be denoted by a - sign in front of the read action. FAM also allows expressing authorizations based on a previous access using so-called done rules. These are essentially facts that are created by the system (FAM) during runtime and reflect the access executed by a user. This helps for example to express the Chinese Wall Policy. The final decision about whether to grant access or deny a request is then resolved by a so-called decision rule. Consider for example the following rule:

$$do(file, s, +a) \leftarrow dercando(file, s, +a) \& \neg dercando(file, s, -a).$$

This specifies that if it can be derived that *s* is allowed to perform action *a* on *file*, and it cannot be derived that *s* is denied to perform action *a* on *file*, then *s* is effectively allowed to perform *a* on *file*. There is no notion of time or temporal dependency between the *done* events. The expression of history-based access control requirements is supported. The history is modeled as a table where each row represents a single access. A row is structured as $(Object, User, Role, Action, Time)$. The history is represented formally by the predicate *done* with a matching list of parameters. It is, however, not clear how the history table is actually updated. The explicit representation of time makes also temporal relations difficult to express at a higher level of abstraction.

2.3.2 Temporal Role-Based Access Control

Role-based Access Control (RBAC) policies regulate the access of users to the information on the basis of the organizational activities and responsibility that users have in a system. In RBAC, a role is defined as the set of privileges associated with a particular position within an organization, or a particular working activity. Authorizations are assigned to roles and roles are assigned to users. The user playing a role is allowed to execute all accesses for which the role is authorized. This mechanism greatly simplifies the management of security policies in that the assignment of roles to users is separated from the assignment of authorizations to roles. Therefore each of these assignments can be manipulated independently.

In [5], a Temporal Role-Based Access Control (TRBAC) model is introduced that addresses temporal constraints for a Role-Based Access Control (RBAC) model. With these constraints one can restrict users to assume roles only at predefined time periods. These time periods are represented by using calenders, a countable set of contiguous intervals, numbered by integers called indexes of the intervals. Examples of calenders are *Hours*, *Days*, *Weeks*, *Months*, and *Years*, where *Hours* is the calendar with the finest granularity. These calenders can be combined to represent more general periodic expressions, denoting periodic instants not necessarily contiguous, such as, for instance, the set of *Mondays* or the set of *The third hour of the first day of each month*. The Role Enabling Base contains temporal constraints on the enabling of roles. Roles can be enabled and disabled at run-time by means of *Run-time Request Expressions* of the form $p : E$ *after* Δt, where $p : E$ is a prioritized event expression, and Δt a duration expression. These requests do not depend on the occurrence of other events and/or the satisfaction of some conditions. The following example involving the medical domain, is from [5]:

Example 4 Let VH (Very High) and H (High) denote priorities with $H \prec VH$. Let *Night-time* and *Day-time* be periodic expressions. Let PE_i denote a periodic event and RT_j denote a role trigger. Then, the following is a Role Enabling Base (REB):

(PE_1) $([1/1/2000, \infty], \textit{Night-time}, VH : \textit{enable doctor-on-night-duty})$
(PE_2) $([1/1/2000, \infty], \textit{Day-time}, VH : \textit{disable doctor-on-night-duty})$
(PE_3) $([1/1/2000, \infty], \textit{Day-time}, VH : \textit{enable doctor-on-day-duty})$
(PE_4) $([1/1/2000, \infty], \textit{Night-time}, VH : \textit{disable doctor-on-day-duty})$
(RT_1) *enable doctor-on-night-duty* $\rightarrow H$: *enable nurse-on-night-duty*
(RT_2) *disable doctor-on-night-duty* $\rightarrow H$: *disable nurse-on-night-duty*
(RT_3) *enable doctor-on-day-duty* $\rightarrow H$: *enable nurse-on-day-duty*
(RT_4) *disable doctor-on-day-duty* $\rightarrow H$: *disable nurse-on-day-duty*
(RT_5) *enable nurse-on-day-duty* $\rightarrow H$: *enable nurse-on-training after* 2
(RT_6) *disable nurse-on-day-duty* $\rightarrow VH$: *disable nurse-on-training*

The periodic events and role triggers in the REB state that the *doctor-on-night-duty* role must be enabled during the night (such constraint is imposed by periodic events PE_1 and PE_2), whereas the role *doctor-on-day-duty* must be enabled during the day (periodic events PE_3 and PE_4). Moreover, role triggers RT_1 and RT_2 state that the role *nurse-on-night-duty* must be enabled whenever the role *doctor-on-night-duty* is. Role triggers RT_3 and RT_4 impose the same constraint for *doctor-on-day-duty* and *nurse-on-day-duty*, respectively. Finally, role triggers RT_5 and RT_6 specify that the role *nurse-on-training* must be enabled only during the daytime when the role *nurse-on-day-duty* is enabled. Moreover, role *nurse-on-training* must be enabled two hours after role *nurse-on-day-duty* is enabled (for instance, because within the first two hours nurses must perform urgent activities and they cannot take care of nurses on training). The following are examples of run-time requests made by a system administrator: *disable nurse-on-training* and *enable emergency-doctor*. The first request has the effect of disabling role nurse-on-training, whereas the second is a request to enable role emergency-doctor.

A system trace is modeled as a sequence of snapshots, where each snapshot corresponds to the current set of events and the status of roles. Unfortunately, because of the expressive power provided by TRBAC, some REB specifications may be ambiguous; that is, they may lead to states where there is no unique way of deciding which roles are enabled. Therefore, a notion of safeness is introduced which guarantees the absence of ambiguities and inconsistencies in the specification. A polynomial algorithm for testing the safety of REB specifications has been given in [5].

Compared with our approach one can see clear differences: TRBAC is mainly concerned with the assignment of roles, whereas our approach is concerned with the assignment of authorizations. Furthermore, TRBAC uses explicit time to model temporal dependencies and incorporates temporal constraints on the role enabling only.

In [35], a generalization of the TRBAC model (GTRBAC) is presented for expressing periodic as well as duration constraints on roles, user-role assignments, and role-permission assignments. In a time interval, activation of a role can further be restricted as a result of numerous activation constraints including cardinality constraints and maximum active duration constraints. The GTRBAC model extends the syntactic structure of the TRBAC model and its event and trigger expressions subsume those of TRBAC. Furthermore, GTRBAC allows expressing role hierarchies and separation of duty (SoD) constraints for specifying fine-grained temporal semantics. In [34], the expressiveness of the constructs provided by the GTBRC is analyzed. It is shown that the constraints-set of GTRBAC is not minimal and that there is a subset of GTRBAC constraints that is sufficient to express all the access constraints that can be expressed using the full set. It is also illustrated that a non-minimal GTRBAC constraint set can provide better flexibility and lower complexity of constraint representation. An approach for the conformance testing of implementations required to enforce access control policies specified using the TRBAC model is proposed in [46]. This uses Timed Input-Output Automata (TIOA) to model the behavior specified by a TRBAC policy. The generated conformance test suite provides complete fault coverage with respect to the proposed fault model for TRBAC specifications.

2.3.3 Usage Control Model

Usage Control (UCON) Models [59, 69], similar to Access Control Models, control and govern the users' access to resources and services that are available in the system. One of the major improvements of UCON over traditional access control models is the continuity of the control and the concept of attribute mutability. The UCON model [59] describes the enforcement of a given policy on a session-based single usage process, i.e., between the start of a usage request and its termination the user can perform a number of actions. The novelty of the approach is that it addresses mutable attributes [56] and the continuity of the enforcement. Mutable attributes are associated with the subjects, objects or the system and are updated as side-effects of usage processes. They can be used for example to count the number of times a resource has been accessed. The continuity of enforcement means that a UCON process can be revoked based on conditions that are expressed in terms of attributes.

The UCON model supports authorization, obligation and conditions. Authorization is concerned with the authorization of a subject to exercise a specific right. Obligations are concerned with actions the user must perform. Conditions are somewhat similar to authorizations, as they also determine the access of a subject — however they depend on a specific class of attributes that are not modified as a part of the system execution. Conditions are described to depend on the environment of the usage process that can for example be influenced by administrative actions.

The UCON model has been first formalized using an extension of the temporal logic of actions (TLA) [41] by Zhang et al. [67, 68]. Here a single usage process is described in the form of a state diagram. System and user actions represent the transitions in the diagram.

UCON policies are then defined as logical formulae that postulate temporal relationships between system and user actions of a single usage process. The formalization, however, makes a strong assumption in that only a single usage process is specified. It is assumed that the time-line is finite, i.e., it starts with the beginning of the single usage request and ends with the subsequent usage request. This makes it difficult to reason about the interactions of several *concurrent* usage requests, or even *sequences* of usage requests, thus complicating the formal analysis of policies. The (side-) effects of a usage process are captured in mutable attributes which are assumed to be persistent over usage processes and can influence subsequent usage control decisions.

A UCON usage process is characterized by the triplet (s, o, r), where s is the subject that exercises its right r on the object o. The usage process can be in one of the following states: *initial*, *requesting*, *denied*, *accessing*, *revoked* or *end*. The current state is described by Zhang et al. as a function *state* mapping from the triplet (s, o, r) to one of these states. A single usage process is defined by the state diagram in Figure 3.

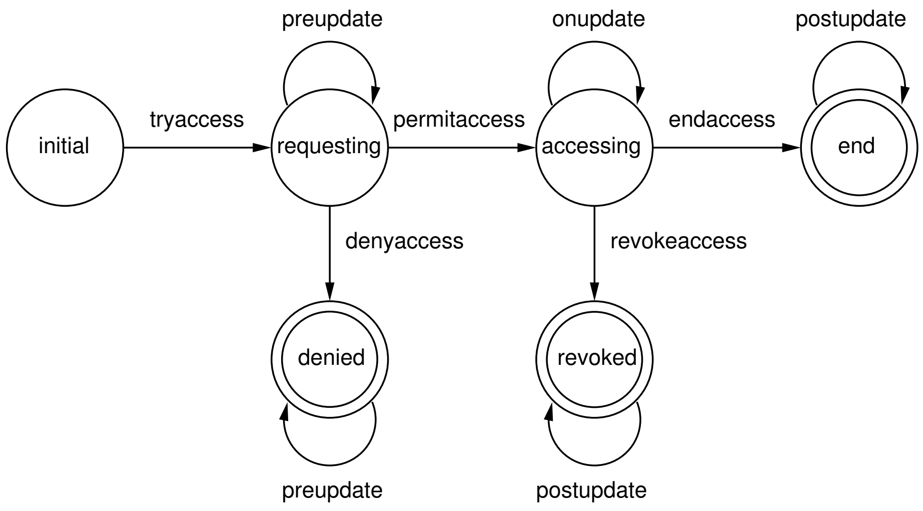


Fig. 3 State diagram adopted from [68]

In the *initial* state the subject s performs the action *tryaccess(s,o,r)* initiating the usage process. The enforcement mechanism, for example a reference monitor (RM), either denies the access (*denyaccess(s,o,r)*) or proceeds by executing actions to update those attributes, which must be updated before the usage process commences. After the RM has updated the relevant attributes (*preupdate(s,o,r)*), it permits the access (*permitaccess(s,o,r)*) and continues to perform all required update actions that must be performed during the ongoing usage process (*onupdate(s,o,r)*). Alternatively the RM may revoke the access if any of the constraints of the UCON model are violated. The subject may end the usage process using the *endaccess(s,o,r)* action. In both cases, the post update actions (*postupdate(s,o,r)*) are performed to change any mutable attributes that require modification.

UCON policies define the enforcement of protection requirements at a relatively low level of abstraction. Our approach has the benefit of addressing the specification of policies at a higher level of abstraction and also providing mechanisms to derive concrete enforcement mechanisms from these specifications.

3 Formal policy language and computational model

We now discuss the policy language that will be used for the specification of history-based access control policy rules. First we will investigate several temporal logics for their suitability to express history-based access control policies. A formal semantic model for the *always-followed-by* operator (used to represent policy rules) will be used for this investigation. We will express the computational model using the chosen logic.

3.1 Choice of Temporal Logic

We first introduce the underlying semantic model to be used in the formalization of history-based access control policies in order to support our choice of left Fusion Logic to express history-based access control policies. We will discuss the suitability of Propositional Interval Temporal Logic (PITL)[49], right Fusion Logic (FL_r)[50], left Fusion Logic (FL_l), Propositional Linear Temporal Logic (PLTL) [45], and Propositional Dynamic Logic (PDL) [24].

3.1.1 Formal semantics of policy rules

The semantic model needs to model sequences of ‘snapshots’ of a system. These sequences represent the *behaviour* of the system. We model these snapshots via a state mapping. A state is a mapping from the set of propositional variables Var to the set of values $\{\mathrm{tt}, \mathrm{ff}\}$. An interval (behavior) σ is a finite sequence of one or more states $\sigma_0\sigma_1\sigma_2\ldots\sigma_{|\sigma|}$, where $|\sigma|$ denotes the length of an interval σ and is equal to the number of states minus 1. Let Σ denote the set of all possible intervals. Let $\sigma = \sigma_0\sigma_1\ldots,\sigma_{|\sigma|}$ be an interval. Then $\sigma_0\ldots\sigma_k$ (where $0 \le k \le |\sigma|$) denotes a *prefix* interval of σ, $\sigma_k\ldots\sigma_{|\sigma|}$ (where $0 \le k \le |\sigma|$) denotes a *suffix* interval of σ and $\sigma_k\ldots\sigma_l$ (where $0 \le k \le l \le |\sigma|$) denotes a *sub*interval of σ.

Let $[\![\,]\!]$ be the “meaning” function from ‘policy rule language’ $\times\,\Sigma$ to $\{\mathrm{tt}, \mathrm{ff}\}$. The formal semantics of the *always-followed-by* operator $Pre \mapsto W$, where Pre is a formula denoting the history and W is an access control variable. The semantics $[\![Pre \mapsto W]\!]_\sigma$ of the *always-followed-by* operator is as follows:

$$\text{for all } k, (0 \le k \le |\sigma| \text{ and } [\![Pre]\!]_{\sigma_0\ldots\sigma_k} = \mathrm{tt}) \text{ implies } [\![W]\!]_{\sigma_k} = \mathrm{tt}.$$

Notice that the implication in the semantics of an individual rule means that W can be true in a state even if Pre does not hold in the prefix of that interval. However, for verification purposes, we need to know the value of W in any state of the interval. This is exactly what the *strong always-followed-by* operator does, it explicitly specifies the conditions under which the access decision is false [33, 62]. This operator will be used in the verification of properties on policy rules (see Section 5) and is defined as

$$Pre \leftrightarrow W \mathrel{\widehat{=}} Pre \mapsto W \wedge \neg Pre \mapsto \neg W.$$

If Pre holds in the prefix interval, then W must hold in the last state of that prefix interval, otherwise W must not hold in that state.

3.1.2 Propositional Interval Temporal Logic

Propositional Interval Temporal Logic (PITL) [49] is a temporal logic with a basic construct for the sequential composition of two formulae as well as an analogue of Kleene star. Within PITL, one can express both finite-state automata and regular expressions. The syntax of PITL is as follows:

PITL formulae	$f ::=$	$p \mid \neg f \mid f_1 \vee f_2 \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$

where skip denotes an interval (sequence) of 2 states, $f_1\,;f_2$ (called 'f_1 chop f_2') denotes the sequential composition of two intervals, and f^* (called 'f chopstar') denotes finite iteration of an interval. PITL assumes the locality principle, i.e., the propositional variables are restricted to be state-based. One evaluates propositional variables over sequences of states by evaluating them in the first state of those sequences. In [48, 50] it was shown that PITL with locality is decidable, though with nonelementary complexity [39] and that PITL without locality is undecidable. In [50] a sound and complete axiom system for PITL with finite time was presented. Let $[\![\ ]\!]$ be the semantic function from PITL Formulae $\times\, \Sigma$ to $\{\mathrm{tt}, \mathrm{ff}\}$. Then, the semantics of PITL is as follows:

- $[\![p]\!]_\sigma = \mathrm{tt}$ iff $\sigma_0(p) = \mathrm{tt}$
- $[\![\neg f]\!]_\sigma = \mathrm{tt}$ iff not $[\![f]\!]_\sigma = \mathrm{tt}$
- $[\![f_1 \vee f_2]\!]_\sigma = \mathrm{tt}$ iff $[\![f_1]\!]_\sigma = \mathrm{tt}$ or $[\![f_2]\!]_\sigma = \mathrm{tt}$
- $[\![\mathsf{skip}]\!]_\sigma = \mathrm{tt}$ iff $|\sigma| = 1$.
- $[\![f_1\,;f_2]\!]_\sigma = \mathrm{tt}$ iff (exists a k, s.t. ($[\![f_1]\!]_{\sigma_0\ldots\sigma_k} = \mathrm{tt}$ and $[\![f_2]\!]_{\sigma_k\ldots\sigma_{|\sigma|}} = \mathrm{tt}$)
- $[\![f^*]\!]_\sigma = \mathrm{tt}$ iff
  (exist $l_0,\ldots,l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
  for all $0 \le i < n, l_i < l_{i+1}$ and $[\![f]\!]_{\sigma_{l_i}\ldots\sigma_{l_{i+1}}} = \mathrm{tt}$)

In order to express a policy rule in PITL we define the following derived operators:

$\mathsf{more} \mathrel{\widehat{=}} \mathsf{skip}\,;\mathsf{true}$	interval with ≥ 2 states.
$\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$	one state interval.
$\diamondsuit\!\!\!\cdot\, f \mathrel{\widehat{=}} f\,;\mathsf{true}$	diamond-i f, i.e., there exists a prefix interval for which f holds.
$\boxdot f \mathrel{\widehat{=}} \neg(\diamondsuit\!\!\!\cdot\,\neg f)$	box-i f, i.e., for all prefix intervals f holds.

It is easy to check via the semantics that a policy rule $Pre \mapsto W$ can be defined as follows

$$Pre \mapsto W \mathrel{\widehat{=}} \boxdot(\neg(Pre\,;((\neg W) \wedge \mathsf{empty})))$$

The *strong always followed by* operator is defined as follows

$$Pre \leftrightarrow W \mathrel{\widehat{=}} \boxdot(\neg(Pre\,;((\neg W) \wedge \mathsf{empty}))) \wedge \boxdot(\neg(\neg Pre\,;(W \wedge \mathsf{empty})))$$

PITL's non-elementary complexity makes the development of efficient PITL-based verification tools difficult. The first verification tool for a subset of PITL is Lite [38], a tableau-based implementation of a decision procedure. Another verification tool is PITL2MONA [22], which is a tool that translates PITL formulae into WS1S and uses MONA as a decision procedure. MONA [20, 36] is an efficient implementation of an automata-based decision procedure for the logic WS1S. In [19] a decision procedure was presented for propositional projection temporal logic with infinite models. DCVALID [54] is a decision procedure for Quantified Discrete-time Duration Calculus (QDDC) [10]. QDDC is closely related to PITL as it supports ‘chop’ and ’chopstar’. Like PITL2MONA, DCVALID uses MONA as its underlying decision procedure.

None of these verification tools scale particularly well, e.g., the MONA based tools introduce too many extra variables in the encoding of the temporal operators, see Example 5.

Example 5
We examine the encoding of the following policy rule

$$\mathsf{true}\,;(A\wedge\mathsf{empty})\,;\mathsf{skip}\,;(B\wedge\mathsf{empty})\leftrightarrow C$$

in both PITL2MONA and DCVALID. This policy rule models an access control (C) that depends on the current (B) and the previous state (A).

Here is the policy rule encoded in PITL2MONA:

```
[i] ( !      (true; A? ; len(1) ; B?   ; (!C)? )),
[i] ( ! ( (! (true; A? ; len(1) ; B?)) ; C?    ))
```

The PITL2MONA tool generates a WS1S formula with 11 variables.

Below is the policy rule encoded in QDDC:

```
discrete;
var A, B, C ;
define policy as
 (! ( true^<A>^(slen=1)^<B>^(<!C>))) &&
(! ( (! ( true^<A>^(slen=1)^<B> ))^ <C>));
infer policy.
```

The DCVALID tool generates a WS1S formula with 13 variables.

Both tools encode temporal operators by formulating the semantics of these operators within WS1S. This semantic encoding introduces extra variables for each temporal operator.

3.1.3 Right Fusion Logic

We next investigate whether right Fusion Logic, a subset of PITL, is a suitable candidate to express policy rules. Right Fusion Logic (FL_r) was first introduced in [50] to prove completeness for Propositional Interval Temporal Logic (PITL). Like PITL, FL augments conventional Propositional Linear Temporal Logic (PLTL) [45] with the chop (fusion) operator. In FL_r the syntax is restricted on the left hand side of the chop operator.

Example 6

– Let ; denote the PITL chop operator. Let F_0 and F_1 be PITL formulae. Then we can write in PITL $\neg F_0\,;\neg F_1$, i.e., the use of negation on both sides of the chop is unrestricted.
– Let $\langle\,\mathsf{test}(\neg P)\,\rangle R$ denote the ‘chop’ between a right fusion logic formula R and a fusion expression $\mathsf{test}(\neg P)$ (restricted syntax), where $\mathsf{test}(\neg P)$ denotes that $\neg P$ holds in an interval with exactly one state. Then $\langle\,\mathsf{test}(\neg P)\,\rangle\neg R$ is a right FL formula but $\langle\,\neg\mathsf{test}(\neg P)\,\rangle\neg R$ is illegal syntax, as one is only allowed to use negations in fusion expressions (left hand side of the chop) if they appear within a state formula $\mathsf{test}(\ )$ or a transition formula $\mathsf{step}(\ )$.

Like PITL, FL$_r$ assumes the locality principle. Despite the restricted syntax, the expressiveness of FL$_r$ is the same as PITL [50], however the complexity is elementary. In [50] a sound and complete axiom system was presented for FL$_r$.

We introduce the four syntactic categories of FL$_r$ in the following table, where

- p is a propositional variable,
- $\bigcirc W$ means that W holds in the next state,
- $\mathsf{test}(W)$ means that W holds in an interval with exactly one state,
- $\mathsf{step}(T)$ a two-state interval satisfying T,
- $E_1 ; E_2$ is a fusion of an interval satisfying fusion expression E_1 that becomes a prefix interval of the resulting interval and an interval satisfying fusion expression E_2 that becomes the suffix interval of the resulting interval such that the last state of the prefix interval is the same as the first state of the suffix interval,
- E^* is the iterative fusion of intervals satisfying fusion expression E, and
- $\langle E \rangle R$ is the fusion of a prefix interval satisfying fusion expression E and a suffix interval satisfying right Fusion Logic formula R.

state formulae	$W ::=$	$\mathsf{true} \mid p \mid W_1 \vee W_2 \mid \neg W$
transition formulae	$T ::=$	$W \mid \bigcirc W \mid T_1 \vee T_2 \mid \neg T$
fusion expressions	$E ::=$	$\mathsf{test}(W) \mid \mathsf{step}(T) \mid E_1 \vee E_2 \mid E_1 ; E_2 \mid E^*$
right fusion logic formulae	$R ::=$	$\mathsf{true} \mid p \mid \neg R \mid R_1 \vee R_2 \mid \langle E \rangle R$

Let $[\![\,]\!]$ be the semantic function from formulae $\times\, \Sigma$ to $\{\mathrm{tt}, \mathrm{ff}\}$ and let $x \in \{W, T, E, R\}$, $y \in \{W, T, R\}$. The semantics of right Fusion Logic is as follows:

- $[\![\mathsf{true}]\!]_\sigma = \mathrm{tt}$
- $[\![p]\!]_\sigma = \mathrm{tt}$ iff $\sigma_0(p) = \mathrm{tt}$
- $[\![x_1 \vee x_2]\!]_\sigma = \mathrm{tt}$ iff $[\![x_1]\!]_\sigma = \mathrm{tt}$ or $[\![x_2]\!]_\sigma = \mathrm{tt}$
- $[\![\neg y]\!]_\sigma = \mathrm{tt}$ iff not $[\![y]\!]_\sigma = \mathrm{tt}$
- $[\![\bigcirc W]\!]_\sigma = \mathrm{tt}$ iff $|\sigma| > 0$ and $[\![W]\!]_{\sigma_1}$
- $[\![\mathsf{test}(W)]\!]_\sigma = \mathrm{tt}$ iff $|\sigma| = 0$ and $[\![W]\!]_{\sigma_0}$
- $[\![\mathsf{step}(T)]\!]_\sigma = \mathrm{tt}$ iff $|\sigma| = 1$ and $[\![T]\!]_{\sigma_0\sigma_1}$
- $[\![E_0 ; E_1]\!]_\sigma = \mathrm{tt}$ iff there exists k s.t. $0 \le k \le |\sigma|$, $[\![E_0]\!]_{\sigma_0 \ldots \sigma_k} = \mathrm{tt}$, and $[\![E_1]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \mathrm{tt}$
- $[\![E^*]\!]_\sigma = \mathrm{tt}$ iff exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
  for all $0 \le i < n, l_i < l_{i+1}$ and $[\![E]\!]_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}} = \mathrm{tt}$
- $[\![\langle E \rangle R]\!]_\sigma = \mathrm{tt}$ iff there exists k s.t. $0 \le k \le |\sigma|$, $[\![E]\!]_{\sigma_0 \ldots \sigma_k} = \mathrm{tt}$, and $[\![R]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \mathrm{tt}$
  Note for $n = 0$, E is not required to hold at the only state σ_0.

The semantics of $\bigcirc W$ and $\mathsf{step}(\mathsf{true}) ; \mathsf{test}(W)$ are the same, however, we wish to keep $\bigcirc W$ as a basic construct because in the decision procedure, a Fusion Logic formula is rewritten into a normal form, containing only $\bigcirc$ as a temporal operator. All propositional variables in the scope of a $\bigcirc$ operator can then be easily replaced by fresh propositional variables so that the resulting formula can be encoded using BDDs [6, 7].

We introduce the following derived right fusion logic formulae in order to express the *always-followed-by* operator.

$\Diamond_r R \mathrel{\widehat{=}} \langle \mathsf{true}_e \rangle R$	sometimes R, i.e., there exists a suffix interval for which R holds.
$\Box_r R \mathrel{\widehat{=}} \neg \Diamond_r(\neg R)$	always R, i.e., for all suffix intervals R holds.
$\Box_r T \mathrel{\widehat{=}} \Box_r \langle \mathsf{step}(T) \rangle \mathsf{true}$	always T, T a transition formula.
$[\, E \,]\, R \mathrel{\widehat{=}} \neg(\langle E \rangle \neg R)$	box-suffix, the dual of the right chop operator.

The *always-followed-by* operator can be defined as $Pre \mapsto W \widehat{=} \big[Pre\big]W$. Pre needs to be a Fusion Expression, i.e., can only contain negation within a transition or state formula. The *strong always-followed-by* operator is then $Pre \leftrightarrow W = \big[Pre\big]W \wedge \big[\neg Pre\big]\neg W$. As $\neg Pre$ needs to be a Fusion Expression, this is illegal syntax for a right Fusion Logic formula. One can move the outer negation to appear inside a state or transition formula of Pre, but this would increase the length of the fusion expression Pre as extra terms are introduced. For example, $\neg(\mathsf{test}(p)\,;\mathsf{step}(\mathsf{true}))$ can be rewritten to $(\mathsf{test}(\neg p)\,;\mathsf{true}_e) \vee \mathsf{empty}_e \vee (\mathsf{step}(\mathsf{true})\,;\mathsf{step}(\mathsf{true})\,;\mathsf{true}_e)$. This and the overhead of moving a negation inside $\mathsf{step}(\ )$ and $\mathsf{test}(\ )$ is the reason for not choosing right Fusion Logic to express policy rules.

3.1.4 Propositional Dynamic Logic (PDL)

We now investigate whether PDL [24] is suitable to express history-based access control rules. The syntax of PDL formulae is as follows: Let a denote primitive program variables, and let p denote propositional variables. Let f denote PDL formulae and let α denote PDL programs.

$$\begin{array}{l} \alpha ::= a \mid \alpha_1\,;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \mid f? \\ f ::= \ p \mid \neg f \mid f_1 \vee f_2 \mid \mathsf{true} \mid \big\langle\, \alpha \,\big\rangle f \end{array}$$

The semantics of PDL maps formulae in a particular state to truth values and all programs to binary state relations. The construct $\big\langle\, \alpha \,\big\rangle f$ is true for a state iff the program α, when started in that state, can possibly terminate in some state in which the formula f is true. Let s denote a state. The semantics of the relevant PDL constructs is as follows:

– $[\![\big\langle\, \alpha \,\big\rangle f]\!]_s = \text{tt}$ iff there exists a state s' s.t. $[\![\alpha]\!]_{(s,s')} = \text{tt}$, and $[\![f]\!]_{s'} = \text{tt}$
– $[\![\alpha_0\,;\alpha_1]\!]_{(s,s')} = \text{tt}$ iff there exists a state s'' s.t. $[\![\alpha_0]\!]_{(s,s'')} = \text{tt}$, and $[\![\alpha_1]\!]_{(s'',s')} = \text{tt}$
– $[\![\alpha^*]\!]_{(s,s')} = \text{tt}$ iff exist states $s_0,\ldots,s_n$ s.t. $s_0 = s$ and $s_n = s'$ and for all $0 \leq i < n, [\![\alpha]\!]_{(s_i,s_{i+1})} = \text{tt}$
– $[\![f?]\!]_{(s,s')} = \text{tt}$ iff $s' = s$ and $[\![f]\!]_s$

PDL is similar to right Fusion Logic because it uses $\big\langle\, \alpha \,\big\rangle f$ where α corresponds to a fusion expression, in particular, a corresponds to a $\mathsf{step}(T)$ fusion expression. However, $f?$ and $\mathsf{test}(W)$ are different in the sense that former allows f to be PDL formula whereas in the latter W can only be a state formula. $f?$ is called a rich test, there is also a poor test where f is restricted to be a state formula, i.e., a poor test is the same as our $\mathsf{test}(W)$. We have already seen that right Fusion Logic is not suitable for expressing history-based access control policies because of the overhead of moving a negation inside $\mathsf{test}(\ )$ and $\mathsf{step}(\ )$.

Sequential Extended Regular Expressions Linear Temporal Logic (SERELTL) [13], the core of the IEEE standard Property Specification Language (PSL) [28], and Regular Linear Temporal Logic (RLTL) [43] are also right Fusion Logics because they have a restricted chop operator that can only have a regular expression on the left hand side. So they are also not suitable for expressing history-based access control policies.

3.1.5 Propositional Linear Temporal Logic (PLTL)

We now investigate whether PLTL [45] is suitable to express history-based access control rules. The syntax of PLTL formulae f is as follows

$$f ::= \mathsf{true} \mid p \mid f_1 \vee f_2 \mid \neg f \mid \bigcirc f \mid \Box f \mid f_1 U f_2$$

The semantics of the until operator is as follows:

– $[\![f_1 U f_2]\!]_\sigma = \text{tt}$ iff there exists a $k : 0 \le k \le |\sigma|, [\![f_2]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \text{tt}$ and for all $j : 0 \le j < k$, $[\![f_1]\!]_{\sigma_j \ldots \sigma_{|\sigma|}} = \text{tt}$

The semantics of $\neg(f_1 U \neg f_2)$ is therefore

– $[\![\neg(f_1 U \neg f_2)]\!]_\sigma = \text{tt}$ iff for all $k : 0 \le k \le |\sigma|, [\![f_2]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \text{tt}$ or not for all $j : 0 \le j < k$, $[\![f_1]\!]_{\sigma_j \ldots \sigma_{|\sigma|}} = \text{tt}$

This is almost the *always-followed-by* operator, i.e., if we restrict f_1 and f_2 to be state formula (no temporal operators) then $\neg(f_1 U \neg f_2)$ corresponds to $(\Box f_1) \mapsto f_2$. One immediately sees that the premise can only be an always type of property. There is no natural way of expressing the sequential composition of, for example, two phases. So it will have difficulty in expressing the 'sequential access' and 'cardinality on history' policy rules of Example 3.

3.1.6 Left Fusion Logic

We will now introduce Left Fusion Logic (FL$_l$). FL$_l$ is similar as FL$_r$, but now the restriction is on the right hand side of the chop. As far as we know, nobody has previously considered this.

The syntax of FL$_l$ is as follows:

left fusion logic formulae	$L ::=$	$\mathsf{true} \mid \mathsf{fin}\,(p) \mid \neg L \mid L_1 \vee L_2 \mid L \langle E \rangle$

where $\mathsf{fin}\,(p)$ means that p holds in the last state, and $L \langle E \rangle$ is the fusion of a prefix interval satisfying left fusion logic formula L and a suffix interval satisfying fusion expression E. The semantics of left Fusion Logic is similar as right Fusion Logic so we only give the semantics for those constructs that are different from FL$_r$.

– $[\![\mathsf{fin}\,(p)]\!]_\sigma = \text{tt}$ iff $\sigma_{|\sigma|}(p) = \text{tt}$
– $[\![L \langle E \rangle]\!]_\sigma = \text{tt}$ iff exists k s.t. $0 \le k \le |\sigma|, [\![L]\!]_{\sigma_0 \ldots \sigma_k} = \text{tt}$, and $[\![E]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \text{tt}$

FL$_l$ is as expressive as PITL. The proof of this is similar as the proof that FL$_r$ is as expressive as PITL [50]. Like FL$_r$, FL$_l$ has elementary complexity. Similarly, one can adapt the sound and complete axiom system of FL$_r$ [50] to obtain a sound and complete axiom system of FL$_l$.

In order to express history-based access control policy rules we introduce the following derived constructs.

$\Diamond_l L \mathrel{\hat{=}} L \langle \mathsf{true}_e \rangle$	diamond-i L, i.e., there exists a prefix interval for which L holds.
$\boxdot_l L \mathrel{\hat{=}} \neg \Diamond_l (\neg L)$	box-i L, i.e., for all prefix intervals L holds.
$L [E] \mathrel{\hat{=}} \neg(\neg L \langle E \rangle)$	box-prefix, the dual of the left chop operator.

Observe that the *always-followed-by* is a prefix type of operator and because of the universal quantification of k we will need a box-prefix operator. It is easy to check via the semantics that a policy rule $Pre \mapsto W$ can be defined as follows

$$Pre \mapsto W \mathrel{\hat{=}} \boxdot_l (\neg Pre\, [\mathsf{test}(\neg W)])$$

As can be seen, we now need to introduce negation already for the *always-followed-by* operator. However, the negation for the access control variable on the right hand side does not cause any trouble as it moves inside the $\mathsf{test}(\ )$. As Pre is a left fusion formula, such a negation is allowed by the syntax.

The *strong always-followed-by* operator is defined as follows

$$\mathit{Pre} \leftrightarrow W \mathrel{\widehat{=}} \boxdot_l(\neg \mathit{Pre}\left[\,\mathsf{test}(\neg W)\,\right]) \wedge \boxdot_l(\mathit{Pre}\left[\,\mathsf{test}(W)\,\right]).$$

3.1.7 Example history-based access control policy rules

The following derived constructs will be used to express the sample history-based access control policy rules of Example 3. First we define some derived fusion expressions:

$\mathsf{len}_e(0) \mathrel{\widehat{=}} \mathsf{test}(\mathsf{true})$ interval of length 0 (one state interval).

$\mathsf{len}_e(n+1) \mathrel{\widehat{=}} \mathsf{step}(\mathsf{true})\,;\mathsf{len}_e(n)$ interval of length $n+1$, for $n \geq 0$.

$\mathsf{true}_e \mathrel{\widehat{=}} \mathsf{step}(\mathsf{true})^*$ finite interval, i.e., any interval of finite length.

$\mathsf{more}_e \mathrel{\widehat{=}} \mathsf{step}(\mathsf{true})\,;\mathsf{true}_e$ non-empty interval, i.e., any interval of length at least one.

$\Diamond_e W \mathrel{\widehat{=}} \mathsf{true}_e\,;\mathsf{test}(W)\,;\mathsf{true}_e$ sometimes W, i.e., there exists a state for which W holds.

$\Box_e W \mathrel{\widehat{=}} \mathsf{step}(W)^*\,;\mathsf{test}(W)$ always W, i.e, for all states W holds.

$n :_e W \mathrel{\widehat{=}} \mathsf{true}_e\,;\mathsf{test}(W)\,;\mathsf{len}_e(n)$ W holds n states from the end.

true_e is a derived fusion expression that is semantically equivalent to the left/right fusion logic formula true. However, true_e cannot be used to define false_e because we have no negation at fusion expression level. One has to define false_e as $\mathsf{test}(\mathsf{false})$. At the fusion logic level one can use true to define false as $\neg\mathsf{true}$ as we have negation.

The fusion expressions derived above, are semantically equivalent to the derived left fusion logic formulae $\mathsf{len}_l(n)$, true, more_l, $\Diamond_l W$, $\Box_l W$ and $n :_l W$, respectively. However, the derived fusion expressions can be used on the right hand side of the left fusion operator as they use only fusion expression operators whereas the corresponding derived left fusion logic formulae cannot be used as they contain the left fusion operator.

Likewise, we need to introduce some derived left fusion logic formulae. Conjunction and implication are defined as usual.

$\mathsf{more}_l \mathrel{\widehat{=}} \mathsf{true}\left\langle\,\mathsf{step}(\mathsf{true})\,\right\rangle$ non-empty interval, i.e., any interval of length at least one.

$\mathsf{empty}_l \mathrel{\widehat{=}} \neg\,\mathsf{more}_l$ empty interval, i.e., any interval of length zero (just one state).

$\mathsf{len}_l(n) \mathrel{\widehat{=}} \begin{cases} \mathsf{empty}_l & n = 0 \\ \mathsf{len}_l(n-1)\left\langle\,\mathsf{step}(\mathsf{true})\,\right\rangle & n > 0 \end{cases}$ interval of length n ($n \geq 0$).

$\Diamond_l E \mathrel{\widehat{=}} \mathsf{true}\left\langle\,E\,\right\rangle$ sometimes E, i.e., there exists a suffix interval for which E holds.

$\Diamond_l W \mathrel{\widehat{=}} \mathsf{true}\left\langle\,\mathsf{test}(W)\,;\mathsf{true}_e\,\right\rangle$ sometimes W, i.e., there exists a suffix interval such that W holds in the first state of that suffix interval.

$\Box_l W \mathrel{\widehat{=}} \neg\,\Diamond_l(\neg W)$ always W, i.e., W holds in the first state of every suffix interval.

$n :_l W \mathrel{\widehat{=}} \mathsf{true}\left\langle\,\mathsf{test}(W)\,;\mathsf{len}_e(n)\,\right\rangle$ W holds n states from the end.

In Example 7 we will express the sample history based access control policy rules of Example 3 in left Fusion Logic.

Example 7 We will only give the formulae.

– Conditional on the current state:
  $0 :_l \mathit{owner}(s,o) \wedge \mathit{account}(o) \mapsto \mathrm{Aut}^+(s,o,\mathbf{withdraw})$

- Conditional on the history:
  $\Diamond_l done(s,o,a) \mapsto \text{Aut}^-(s,o,a)$
- Conditional on the history:
  $\Diamond_l done(s_2,o,a) \wedge s_1 \neq s_2 \mapsto \text{Aut}^-(s_1,o,a)$
- Conditional on the history:
  $\Diamond_l(\mathsf{test}(\text{signin}_{\text{hj}})\,;\Box_e(\neg\,\text{signout}_{\text{hj}})) \mapsto \text{Aut}^+(\mathbf{hj},\mathbf{door},\mathbf{open})$
- Invariant:
  $\Box_l \neg\, bankrupt(s) \mapsto \text{Aut}^+(s,\mathbf{loan},\mathbf{take})$
- Choice:
  $$\left(\begin{array}{l} s_1 \neq s_2 \wedge parent(s_1,s) \wedge parent(s_2,s) \wedge \\ ((age\_less\_10(s) \wedge \\ \quad \Diamond_l done(s_1,o,\mathbf{consent}(a)) \wedge \Diamond_l done(s_2,o,\mathbf{consent}(a))) \vee \\ (\neg\, age\_less\_10(s) \wedge \\ \quad (\Diamond_l done(s_1,o,\mathbf{consent}(a)) \vee \Diamond_l done(s_2,o,\mathbf{consent}(a))))) \end{array}\right) \mapsto \text{Aut}^+(s,o,a)$$
- Collaboration:
  $5 :_l \Diamond_l req(s_1,\mathbf{door},\mathbf{open}) \wedge \Diamond_l req(s_2,\mathbf{door},\mathbf{open}) \wedge s_1 \neq s_2 \mapsto \text{Aut}^+(s,\mathbf{door},\mathbf{open})$
- Sequential access:
  $$\left(\begin{array}{l} \neg(\Diamond_l(\mathsf{test}(done(s,\mathbf{invoice},\mathbf{receive}));\\ \qquad \Diamond_e done(s_a,\mathbf{invoice},\mathbf{authorize}))) \wedge \\ 0 :_l role(s_a,\mathbf{accountant}) \end{array}\right) \mapsto \text{Aut}^-(s,\mathbf{bank},\mathbf{pay}(\mathbf{invoice}))$$
- Counting states in the history:
  $(\bigvee_{i=0}^{3} i :_l deleg(s,s',o,a)) \wedge 0 :_l \text{Aut}^+(s,o,a) \mapsto \text{Aut}^+(s',o,a)$
- Cardinality on history
  $\Diamond_l((\mathsf{test}(req(s,o,a))\,;\Diamond_e req(s,o,a))^*) \mapsto \text{Aut}^+(s,o,a)$

3.2 Expressing the Computational model in left Fusion Logic

The formalization of the user, reference monitor and system processes that are part of the computational model (see Figure 2) for policy based management is as follows:

- User process: Let S be the set of subjects. Then the user process consists of all the subject processes in parallel, i.e., $\bigwedge_{s\in S} User(s)$.
  Let $UI(s,o,a)$ denote

  $$\begin{array}{l} \mathsf{step}(u\_idle_s \wedge \neg\, req(s,o,a))^*;\\ \mathsf{test}(u\_idle_s \wedge req(s,o,a))\,;\mathsf{step}(\mathsf{true}) \end{array}$$

  This represents that the subject process s is waiting in the *idle* state, and that on the generation of the *request* event it moves to the next state. We use propositional variables to describe that a process is in a particular state, i.e., u_idle_s has the value true when subject process s is in the *idle* state. Furthermore, we will use propositional variables to describe events, i.e., $req(s,o,a)$ is true when subject process s sends a request to the reference monitor to perform action a on object o.
  Let $UW_0(s,o,a)$ denote

  $$\begin{array}{l} \mathsf{step}(u\_wait_s \wedge \neg\, exec(s,o,a) \wedge \neg\, deny(s,o,a))^*;\\ \mathsf{test}(u\_wait_s \wedge exec(s,o,a) \wedge \neg\, deny(s,o,a))\,;\mathsf{step}(\mathsf{true}) \end{array}$$

  representing that the subject process is in the *wait* state and waiting for the *exec* event, after which it moves to the next state.

Let $UW_1(s,o,a)$ denote

$$\mathsf{step}(u\_wait_s \wedge \neg exec(s,o,a) \wedge \neg deny(s,o,a))^*;$$
$$\mathsf{test}(u\_wait_s \wedge \neg exec(s,o,a) \wedge deny(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject process is in the *wait* state and waiting for the *deny* event, after which it moves to the next state.
Let $UA(s,o,a)$ denote

$$\mathsf{step}(access_s \wedge \neg done(s,o,a))^*;$$
$$\mathsf{test}(access_s \wedge done(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject process is in the *access* state and waiting for the *done* event, after which it moves to the next state.
Then each subject process $User(s)$ is defined as follows:

$$\mathsf{empty}_l \left\langle (\bigvee_{o\in O, a\in A} (UI(s,o,a)\,;(UW_0(s,o,a)\,;UA(s,o,a) \vee UW_1(s,o,a))))^*\,;\mathsf{true}_e \right\rangle.$$

Each subject process corresponds to the user Statechart in Figure 2. The ;true_e after the * is needed to "pad" a subject process with states so that it has the same length as other subject processes when put in parallel with those processes.

– Reference monitor process: Associated with each subject process is a subject reference monitor process for that subject. Accordingly the reference monitor process consists of those subject reference monitor processes in parallel, i.e., $\bigwedge_{s\in S} RM(s)$.
Let $RI(s,o,a)$ denote

$$\mathsf{step}(rm\_idle_s \wedge \neg req(s,o,a))^*;$$
$$\mathsf{test}(rm\_idle_s \wedge req(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject reference monitor process is in the *idle* state and waiting for the *request* event, after which it moves to the next state.
Let $RP_0(s,o,a)$ denote

$$\mathsf{test}(process_s \wedge \mathrm{Aut}(s,o,a) \wedge permit(s,o,a) \wedge \neg deny(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject reference monitor process is in the *process* state and if the policy grants access ($\mathrm{Aut}(s,o,a)$ is true), it generates the *permit* event and moves to the next state. $\mathrm{Aut}(s,o,a)$ is again a propositional variable.
Let $RP_1(s,o,a)$ denote

$$\mathsf{test}(process_s \wedge \neg \mathrm{Aut}(s,o,a) \wedge \neg permit(s,o,a) \wedge deny(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject reference monitor process is in the *process* state and if the policy denies access ($\mathrm{Aut}(s,o,a)$ is false), it generates the *deny* event and moves to the next state.
Let $RW(s,o,a)$ denote

$$\mathsf{step}(r\_wait_s \wedge \neg done(s,o,a))^*;$$
$$\mathsf{test}(r\_wait_s \wedge done(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

representing that the subject reference monitor process is in the *wait* state and waiting for the *done* event, after which it moves to the next state.
Then each subject reference monitor process $RM(s)$ is defined as follows:

$$\mathsf{empty}_l \left\langle (\bigvee_{o\in O, a\in A} (RI(s,o,a)\,;(RP_0(s,o,a)\,;RW(s,o,a) \vee RP_1(s,o,a))))^*\,;\mathsf{true}_e \right\rangle.$$

– System process: Similarly with each subject process there is a corresponding subject system process. The system process consists of those subject system processes in parallel, i.e., $\bigwedge_{s\in S} Sys(s)$.
  Let $SI(s,o,a)$ denote

$$\mathsf{step}(s\_idle_s \land \neg permit(s,o,a) \land \neg exec(s,o,a))^*;\\ \mathsf{test}(s\_idle_s \land permit(s,o,a) \land exec(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

  representing that the subject system process is in the *idle* state and waiting for the *permit* event, after which it generates the *exec* event and moves to the next state.
  Let $SE(s,o,a)$ denote

$$\mathsf{step}(execute_s \land \neg done(s,o,a))^*;\\ \mathsf{test}(execute_s \land done(s,o,a))\,;\mathsf{step}(\mathsf{true})$$

  representing that the subject system process is waiting in the *exec* state, on the generation of the *done* event it moves to the next state.
  Then each subject system process $Sys(s)$ is defined as follows:

$$\mathsf{empty}_l \left\langle (\bigvee_{o\in O, a\in A} (SI(s,o,a)\,;SE(s,o,a)))^*\,;\mathsf{true}_e \right\rangle.$$

The computational model is as follows:

$$P \land \bigwedge_{s\in S}(User(s) \land RM(s) \land Sys(s)),$$

where P denotes the history-based access control policy. In the next section we show how to express a policy P in Fusion Logic.

4 Verification and Enforcement of policy rules

In this section, we show how the verification of properties on policy rules can be translated into a BDD-based decision procedure for Fusion Logic. For the specification of the policy rules and properties one uses FL_l. Internally, the decision procedure translates FL_l formula into FL_r formula using time reversal. Furthermore, we show how this decision procedure can be adapted to act as an enforcer.

Model-checking techniques for the analysis of policies has been explored for policy models e.g., [60, 70], or for formalizations of XACML [53] in [37]. We did not use standard off the shelf model-checkers like [12, 25] as we wanted to adapt the decision procedure so that it can also be used for the enforcement of policy rules.

The problem of finding an enforcer is an instance of Church's Problem [11, 65] where one asks for an automaton that realizes the transformation of an input sequence into an output sequence such that a requirement on those sequences is satisfied. The solution of Church's Problem is an automaton that produces output depending on the user input in such a way that the requirements are satisfied. Our enforcer is, likewise, an automaton generated from the requirements on the input and output sequence.

The BDDs that are constructed during the enforcement of a policy encode an automaton, similar to the security automaton of [61]. BDDs have been also used for computing the similarity of XAMCL policies as part of the EXAM project, based on the notion of Rule Set Similarity, first explored in [58]. More recent work by Hu [26] also uses BDDs to detect conflicts and policy anomalies in XACML policy specifications. A similar approach as also been used to reason about Spectrum Access Policies in e.g. [3] an the computation of opportunity constraints. However, in these approaches only static policies are considered. The dependence on the system-history, as considered in this paper, is making policies dynamic and adaptable to the system context. The verification of properties of these policies and the generation of a corresponding enforcement mechanism, is a more complex problem as the system trace must be taken into account.

4.1 Verification

A policy rule is a left Fusion Logic formula and since we base our verification technique on BDDs[6, 7], we must find a way to translate a left Fusion Logic formula into a Boolean function. We base our decision procedure for left Fusion Logic on the decision procedure for right Fusion Logic presented in [51]. The difference between the one presented in [51] and the one presented in this paper is that we have now as input a FL_l formula which is first translated into a FL_r formula. Furthermore we use a more efficient reduction step where the right Fusion Logic formula is transformed into an automaton. This reduction step is similar to the reduction technique presented by Fisher et al. in [21]. In Section 4.1.3 we will compare the three reduction techniques.

First we recall some basic definitions. A Fusion Logic formula F is *satisfiable* if and only if there exists an interval σ such that $[\![F]\!]_\sigma = \mathrm{tt}$. A *decision procedure* for Fusion Logic has as input a Fusion Logic formula F, and as output a decision whether F is satisfiable or not. The decision procedure will also generate a satisfying interval in the case the formula is satisfiable. A Fusion Logic formula F is *valid* if and only if for all intervals σ, $[\![F]\!]_\sigma = \mathrm{tt}$. A decision procedure for Fusion Logic can also be used to check validity of a formula as we can express validity of a formula in terms of satisfiability: F is *not* valid if and only if $\neg F$ is satisfiable, i.e., the satisfying interval for $\neg F$ will represent a *counter example* for F's validity. Therefore, F is valid if and only if $\neg F$ is *not* satisfiable.

The decision procedure for left Fusion Logic consists of three steps:

1. Time reversal step. Transform a FL_l formula into a FL_r formula using time reversal. Continue with the next step.
2. Reduction Step. Transform a right Fusion Logic formula into a formula of the form $init \wedge \Box_r I$, where $init$ is a state formula and I is a transition formula. Continue with the next step.
3. BDD Step. Transform $init \wedge \Box_r I$ into a BDD-based satisfiability problem.

In the following, we will discuss each of these steps in more detail.

4.1.1 Time Reversal

In [52], time reversal was used by Moszkowski to verify certain properties expressed in PITL. Here it is used to rewrite left fusion formulae into right fusion logic formulae as part of the decision procedure.

Time reversal is related to mirror images (see Prior [57]) used for temporal logics to obtain a rule for past-time operators from an analogous one for future-time operators by means of time symmetry.

Let formula F^r denote the *time reversed* version of F. Let the time reversed interval of a finite interval σ be denoted by $reverse(\sigma)$ and be defined as $reverse(\sigma_0 \dots \sigma_{|\sigma|}) \mathrel{\widehat{=}} \sigma_{|\sigma|} \dots \sigma_0$. The semantics of time reversal is defined as

$$[\![F^r]\!]_\sigma = \text{tt iff } [\![F]\!]_{reverse(\sigma)} = \text{tt}$$

As the semantics of left/right Fusion Logic is defined over finite intervals we can use this definition for time reversal.

In Table 1, we list the rewrite rules for transforming left fusion logic formulae into right fusion logic formulae.

left fusion formulae	fusion expressions	transitions
$((L\langle E \rangle))^r = \langle E^r \rangle L^r$	$(\mathsf{test}(W))^r = \mathsf{test}(W)$	$(\bigcirc W)^r = W$
$(\mathsf{fin}\,(p))^r = p$	$(\mathsf{step}(T))^r = \mathsf{step}(T^r)$	$W^r = \bigcirc W$
$\mathsf{true}^r = \mathsf{true}$	$(E_1 \vee E_2)^r = E_1^r \vee E_2^r$	$(T_1 \vee T_2)^r = T_1^r \vee T_2^r$
$(\neg L)^r = \neg(L^r)$	$(E_1 ; E_2)^r = E_2^r ; E_1^r$	$(\neg T)^r = \neg(T^r)$
$(L_1 \vee L_2)^r = L_1^r \vee L_2^r$	$(E^*)^r = (E^r)^*$	

Table 1 Time reversal rules

Lemma 1 *Let L be a left Fusion Logic formula. Then, L^r is a right Fusion Logic formula.*

Proof Use structural induction on L and the rewrite rules in Table 1. Details omitted.

Lemma 2 allows for a decision procedure for right fusion logic formulae to be used for left fusion logic formulae, i.e., time reverse the left fusion logic formula L into a right fusion logic formula L^r and then determine the satisfiability of L^r using the decision procedure. If satisfiable, then there exists a σ such that $[\![L^r]\!]_\sigma = \text{tt}$, and thus, using time reversal, $[\![L]\!]_{reverse(\sigma)} = \text{tt}$.

Lemma 2 (Left satisfiability as right satisfiability) *Let L be a left fusion logic formula. Then, $[\![L]\!]_\sigma = tt$ iff $[\![L^r]\!]_{reverse(\sigma)} = tt$.*

Proof $[\![L]\!]_\sigma = \text{tt}$ iff, using the fact that reversing an interval twice is the interval itself, $[\![L]\!]_{reverse(reverse(\sigma))} = \text{tt}$. Using the definition of the reversal operator, where one substitutes for σ, $reverse(\sigma)$, we get $[\![L^r]\!]_{reverse(\sigma)} = \text{tt}$. QED

4.1.2 Reduction Step

In this step, we will introduce a mechanism to rewrite a right fusion logic formula into an automaton specified by an initial state and a transition relation. Automata are only indirectly introduced via a corresponding normal form. Reduction rules will be introduced that rewrite a right fusion logic formula into this form.

The reduction step will transform a right fusion logic formula R, possibly introducing dependent variables, into the following equivalent reduced form $init \wedge \Box_r I$, where $init \mathrel{\widehat{=}} \mathscr{R}_0'(R)$ and $I \mathrel{\widehat{=}} \mathscr{R}_0(R)$. For $k \in \{0,1\}$, $\mathscr{R}_k(R)$ is a transition formula, $\mathscr{R}_0'(R)$ is a state formula, and $\mathscr{R}_1'(R)$ is a transition formula. Let X, X_1, and X_2 denote *non-state* right fusion logic formulae and W a *state* formula. Then the definitions of transition formulae $\mathscr{R}_0(R)$, $\mathscr{R}_1(R)$ and $\mathscr{R}_1'(R)$, and state formula $\mathscr{R}_0'(R)$ are given in Table 2. Some of the $\mathscr{R}_0(R)$ transformations call $\mathscr{R}_0'(R)$, $\mathscr{R}_1(R)$, and $\mathscr{R}_1'(R)$ transformations, and some of the $\mathscr{R}_1(R)$ transformations call

$\mathscr{R}_0(R)$ transformations, and some $\mathscr{R}'_1(R)$ transformations call $\mathscr{R}'_0(R)$ transformations. It is worth pointing out that only the $\mathsf{step}(T)$ and E^* cases will introduce a dependent variable (denoted by respectively $r_{\langle \mathsf{step}(T) \rangle X}$ and $r_{\langle E^* \rangle X}$).

R	$\mathscr{R}_0(R)$	$\mathscr{R}_1(R)$
W	true	true
$\langle \mathsf{test}(W) \rangle X$	$\mathscr{R}_0(X)$	$\mathscr{R}_1(X)$
$\langle \mathsf{step}(T) \rangle X$	$(r_{\langle \mathsf{step}(T) \rangle X} \equiv (T \wedge \bigcirc\mathscr{R}'_0(X))) \wedge \mathscr{R}_0(X)$	$\mathscr{R}_0(X)$
$\langle E_1 \vee E_2 \rangle X$	$\mathscr{R}_0(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$	$\mathscr{R}_1(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$
$\langle E_1 ; E_2 \rangle X$	$\mathscr{R}_0(\langle E_1 \rangle \langle E_2 \rangle X)$	$\mathscr{R}_1(\langle E_1 \rangle \langle E_2 \rangle X)$
$\langle E^* \rangle X$	$(r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1)) \wedge \mathscr{R}_1(X_1)$	$(r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1)) \wedge \mathscr{R}_1(X_1)$
	where X_1 is $X \vee \langle c(E) \rangle r_{\langle E^* \rangle X}$	where X_1 is $X \vee \langle c(E) \rangle r_{\langle E^* \rangle X}$
$\neg X$	$\mathscr{R}_0(X)$	$\mathscr{R}_1(X)$
$X_1 \vee X_2$	$\mathscr{R}_0(X_1) \wedge \mathscr{R}_0(X_2)$	$\mathscr{R}_1(X_1) \wedge \mathscr{R}_1(X_2)$

R	$\mathscr{R}'_0(R)$	$\mathscr{R}'_1(R)$
W	W	W
$\langle \mathsf{test}(W) \rangle X$	$W \wedge \mathscr{R}'_0(X)$	$W \wedge \mathscr{R}'_1(X)$
$\langle \mathsf{step}(T) \rangle X$	$r_{\langle \mathsf{step}(T) \rangle X}$	$T \wedge \bigcirc\mathscr{R}'_0(X)$
$\langle E_1 \vee E_2 \rangle X$	$\mathscr{R}'_0(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$	$\mathscr{R}'_1(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$
$\langle E_1 ; E_2 \rangle X$	$\mathscr{R}'_0(\langle E_1 \rangle \langle E_2 \rangle X)$	$\mathscr{R}'_1(\langle E_1 \rangle \langle E_2 \rangle X)$
$\langle E^* \rangle X$	$r_{\langle E^* \rangle X}$	$r_{\langle E^* \rangle X}$
$\neg X$	$\neg\mathscr{R}'_0(X)$	$\neg\mathscr{R}'_1(X)$
$X_1 \vee X_2$	$\mathscr{R}'_0(X_1) \vee \mathscr{R}'_0(X_2)$	$\mathscr{R}'_1(X_1) \vee \mathscr{R}'_1(X_2)$

E	$c(E)$
$\mathsf{test}(W)$	$\mathsf{test}(\neg\mathsf{true})$
$\mathsf{step}(T)$	$\mathsf{step}(T)$
$E_1 \vee E_2$	$c(E_1) \vee c(E_2)$
$E_1 ; E_2$	$c(E_1) ; E_2 \vee E_1 ; c(E_2)$
E^*	$c(E) ; E^*$

Table 2 Definition of transition formulae $\mathscr{R}_0(R)$, $\mathscr{R}_1(R)$, and $\mathscr{R}'_1(R)$, state formula $\mathscr{R}'_0(R)$, and function $c(E)$

The disjunction of two fusion logic formulae is turned into a conjunction for the transition relation $\mathscr{R}_0(R)$ because the 'choice' is made in the initial state $\mathscr{R}'_0(R)$ and in the transition relation $\mathscr{R}'_1(R)$. Example 8 illustrates this.

The construction of the invariant for $\langle E^* \rangle W$ is a bit more involved because the fusion expression E could be valid on some empty intervals. Therefore a function c is introduced that strengthens an arbitrary fusion expression E to $c(E)$ that holds over a non empty interval, i.e., $c(E) \equiv E \wedge \mathsf{more}_e$. Because $\langle E^* \rangle W \equiv \langle c(E)^* \rangle W$ holds, this strengthening is justified. The function $c(E)$ is defined in Table 2.

The reduction rules in Table 2 are similar to the (partial) derivative of regular expressions used in finite automata construction [2, 8]. In our decision procedure we construct an automaton where the transitions have no labels, i.e., variables reside in the state and make up the alphabet of the automata. In the automata constructed using (partial) derivatives, transitions are labeled by a letter from the alphabet.

The reduction rules in Table 2 enable us to reduce a right fusion logic formula into a formula of the form $init \wedge \Box_r \bigwedge_{i=1}^{k}(r_{X_i} \equiv t_i)$, where $init$ is a state formula, r_{X_i} is a dependent variable and t_i is a transition formula. Example 8 gives an example of reduction of a right fusion logic formula.

Example 8
Consider the following right fusion logic formula X:

$$\langle\, \mathsf{step}(A)^* \,\rangle\, (B \vee C) \vee \langle\, \mathsf{step}(A)\,; \mathsf{test}(B) \,\rangle\, D$$

X represents an interval where A holds in each state except possibly in the last state where either B or C holds or it is an interval with two states where in the first state A holds and in the second state B and D holds.

We want to transform X into an $init \wedge \Box_r I$ formula using the reduction rules in Table 2.

– We know that $init \mathrel{\widehat{=}} \mathscr{R}'_0(X)$. Using the reduction rule for disjunction we get, $\mathscr{R}'_0(X) = \mathscr{R}'_0(X_1) \vee \mathscr{R}'_0(X_2)$, where $X_1 \mathrel{\widehat{=}} \langle\, \mathsf{step}(A)^* \,\rangle\, (B \vee C)$ and $X_2 \mathrel{\widehat{=}} \langle\, \mathsf{step}(A)\,; \mathsf{test}(B) \,\rangle\, D$. Using the reduction rule for 'chopstar' we get, $\mathscr{R}'_0(X_1) = r_{X_1}$. Using the reduction rule for 'chop' we get, $\mathscr{R}'_0(X_2) = \mathscr{R}'_0(X_3)$, where $X_3 \mathrel{\widehat{=}} \langle\, \mathsf{step}(A) \,\rangle \langle\, \mathsf{test}(B) \,\rangle\, D$. Using the reduction rule for 'step' we get, $\mathscr{R}'_0(X_3) = r_{X_3}$. So

$$init = r_{X_1} \vee r_{X_3}.$$

– We now proceed with invariant I. We know that $I \mathrel{\widehat{=}} \mathscr{R}_0(X)$. Using the reduction rule for 'disjunction' we get, $\mathscr{R}_0(X) = \mathscr{R}_0(X_1) \wedge \mathscr{R}_0(X_2)$, where X_1 and X_2 are defined as above. Using the reduction rule for 'chopstar' we get, $\mathscr{R}_0(X_1) = (r_{X_1} \equiv \mathscr{R}'_1(X_4)) \wedge \mathscr{R}_1(X_4)$, where $X_4 \mathrel{\widehat{=}} (B \vee C) \vee \langle\, c(\mathsf{step}(A)) \,\rangle\, r_{X_1})$. Using the definition of $c(\ )$ for 'step' we get, $X_4 = (B \vee C) \vee \langle\, \mathsf{step}(A) \,\rangle\, r_{X_1})$. Using the definition of 'disjunction' we get, $\mathscr{R}'_1(X_4) = \mathscr{R}'_1(B \vee C) \vee \mathscr{R}'_1(\langle\, \mathsf{step}(A) \,\rangle\, r_{X_1})$. Using the reduction step for 'state formula' and 'step' we get, $\mathscr{R}'_1(X_4) = (B \vee C) \vee (A \wedge \bigcirc \mathscr{R}'_0(r_{X_1}))$. Using the reduction step for 'state formula' we get, $\mathscr{R}'_1(X_4) = (B \vee C) \vee (A \wedge \bigcirc r_{X_1})$. Reduction of $\mathscr{R}'_1(X_4)$ is now complete, and so we now proceed with $\mathscr{R}_1(X_4)$. Using the reduction step for 'disjunction' we get, $\mathscr{R}_1(X_4) = \mathscr{R}_1(B \vee C) \wedge \mathscr{R}_1(\langle\, \mathsf{step}(A) \,\rangle\, r_{X_1})$. Using the reduction step for 'state formula' and 'step' we get, $\mathscr{R}_1(X_4) = \mathsf{true} \wedge \mathscr{R}_0(r_{X_1})$. Using the reduction step for 'state formula' we get, $\mathscr{R}_1(X_4) = \mathsf{true} \wedge \mathsf{true}$. Reduction $\mathscr{R}_1(X_4)$ is now complete and therefore $\mathscr{R}_0(X_1) = (r_{X_1} \equiv (B \vee C) \vee (A \wedge \bigcirc r_{X_1}))$. We proceed with $\mathscr{R}_0(X_2)$. Using the reduction step for 'chop' we get, $\mathscr{R}_0(X_2) = \mathscr{R}_0(X_3)$, where X_3 is as above. Using the reduction rule for 'step' we get, $\mathscr{R}_0(X_3) = (r_{X_3} \equiv A \wedge \bigcirc \mathscr{R}'_0(X_5)) \wedge \mathscr{R}_0(X_5)$, where $X_5 \mathrel{\widehat{=}} \langle\, \mathsf{test}(B) \,\rangle\, D$. Using the reduction rule for 'test' we get, $\mathscr{R}'_0(X_5) = B \wedge \mathscr{R}'_0(D)$. Using the reduction rule for 'state formula' we get, $\mathscr{R}'_0(X_5) = B \wedge D$. $\mathscr{R}'_0(X_5)$ is now complete, and we proceed with $\mathscr{R}_0(X_5)$. Using the reduction rule for 'test' we get, $\mathscr{R}_0(X_5) = \mathscr{R}_0(D)$. Using the reduction rule for 'state formula' we get, $\mathscr{R}_0(X_5) = \mathsf{true}$. Reduction $\mathscr{R}_0(X_5)$ is now complete and therefore $\mathscr{R}_0(X_2) = (r_{X_3} \equiv A \wedge \bigcirc(B \wedge D))$. The invariant I is therefore

$$I = (r_{X_1} \equiv (B \vee C) \vee (A \wedge \bigcirc r_{X_1})) \wedge (r_{X_3} \equiv A \wedge \bigcirc(B \wedge D))$$

The following example shows how dependent variables are used to count states.

Example 9
Let us consider the following right fusion logic formula $R \mathrel{\widehat{=}} \langle\, \mathsf{step}(A)^* \,\rangle\, B \wedge \langle\, \mathsf{len}_e(4) \,\rangle\, \mathsf{empty}_r$, where empty_r denotes the formula $\neg \langle\, \mathsf{step}(\mathsf{true}) \,\rangle\, \mathsf{true}$. R represents a 5 state interval, where in the first 4 states A holds and in the last state B holds. Given the following right fusion logic formulae:

$X_1 \mathrel{\widehat{=}} \langle\, \mathsf{step}(A)^* \,\rangle\, B \qquad X_4 \mathrel{\widehat{=}} \langle\, \mathsf{len}_e(2) \,\rangle\, \mathsf{empty}_r$
$X_2 \mathrel{\widehat{=}} \langle\, \mathsf{len}_e(4) \,\rangle\, \mathsf{empty}_r \quad X_5 \mathrel{\widehat{=}} \langle\, \mathsf{len}_e(1) \,\rangle\, \mathsf{empty}_r$
$X_3 \mathrel{\widehat{=}} \langle\, \mathsf{len}_e(3) \,\rangle\, \mathsf{empty}_r \quad X_6 \mathrel{\widehat{=}} \langle\, \mathsf{step}(\mathsf{true}) \,\rangle\, \mathsf{true}$

Then R is reduced to $init \wedge \Box_r I$, where $init = r_{X_1} \wedge r_{X_2}$ and

$$\begin{aligned} I = {} & (r_{X_1} \equiv (B \vee (A \wedge \bigcirc r_{X_1}))) \wedge (r_{X_2} \equiv \bigcirc r_{X_3}) \wedge (r_{X_3} \equiv \bigcirc r_{x_4}) \wedge \\ & (r_{X_4} \equiv \bigcirc r_{x_5}) \wedge (r_{X_5} \equiv \neg \bigcirc r_{X_6}) \wedge (r_{X_6} \equiv \mathsf{true}) \end{aligned}$$

The states of the interval representing the formula are then as follows:

$$\begin{array}{ccccc} r_{X_1}, r_{X_2} & r_{X_1}, r_{X_3} & r_{X_1}, r_{X_4} & r_{X_1}, r_{X_5} & r_{X_1}, \neg r_{X_6} \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ A & A & A & A & B \end{array}$$

Theorem 1 is the logical underpinning which explains why the reduction rules in Table 2 give us the desired result.

Theorem 1 *Let R be a right fusion logic formula and $dep(R)$ be the dependent variables introduced by $\mathscr{R}'_k(R)$ and $\mathscr{R}_k(R)$ ($k = 0, 1$). Then,*

$$R \equiv \exists dep(R) \bullet (\mathscr{R}'_k(R) \wedge \Box_r \mathscr{R}_k(R))$$

is valid.

The proof of Theorem 1 uses the following two lemmas.

Lemma 3 *Let R be a right fusion logic formula and $dep(R)$ be the dependent variables introduced by $\mathscr{R}'_k(R)$ and $\mathscr{R}_k(R)$ ($k = 0, 1$). Then the formula below is valid:*

$$\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$$

Lemma 4 *Let R be a right fusion logic formula. Then the next implication is valid:*

$$\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$$

The proofs of Lemma 3 and 4 are given in Appendix A and use structural induction on the syntax of a right Fusion Logic formula. The proof of Theorem 1 is as follows:

Proof R is equal to, using Lemma 3, $R \wedge \exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$. Moving R inside the existential quantification gives us $\exists dep(R) \bullet (R \wedge \Box_r \mathscr{R}_k(R))$. Using Lemma 4 gives us $\exists dep(R) \bullet (R \wedge \Box_r \mathscr{R}_k(R) \wedge \mathscr{R}'_k(R) \equiv R)$. Using $\mathscr{R}'_k(R) \equiv R$ gives us $\exists dep(R) \bullet (\mathscr{R}'_k(R) \wedge \Box_r \mathscr{R}_k(R))$, the desired result. QED

4.1.3 Comparison with other reduction techniques

We will compare our reduction step with the one presented in [51] and the one presented by Fisher et al. in [21].

Reduction technique of Moszkowski [51]: Again a right Fusion Logic formula R is transformed into an equivalent reduced form $r_{|\mathscr{R}(R)|} \wedge \Box_r \mathscr{R}(R)$, where $|\mathscr{R}(R)|$ denotes the number of distinct dependent variables in $\mathscr{R}(R)$. Let $I \uparrow k$ denote the formula I with the subscripts of all dependent variables increased by k, i.e., r_i becomes r_{i+k}. Let $I[q \leftarrow r_k]$ denote the formula with all occurrences of q replaced by r_k. The reduction rules are presented in Table 3, where X, X_1, and X_2 denote non-state formulae and W a state formula. As can be seen, $\mathscr{R}(R)$ is of the form $\bigwedge_{i=1}^{k} (r_i \equiv t_i)$, where r_i is a dependent variable and t_i is a transition formula.

Notice that the reduction technique introduces a dependent variable for each right Fusion Logic operator. In Example 10 we reduce the same formula as Example 8.

R	$\mathscr{R}(R)$
W	$r_1 \equiv W$
$\langle\, \mathsf{test}(W') \,\rangle W$	$r_1 \equiv (W \wedge W')$
$\langle\, \mathsf{step}(T) \,\rangle W$	$r_1 \equiv (T \wedge \bigcirc W)$
$\langle\, E_1 \vee E_2 \,\rangle W$	$\mathscr{R}(\langle\, E_1 \,\rangle W \vee \langle\, E_2 \,\rangle W)$
$\langle\, E_1 ; E_2 \,\rangle W$	$\mathscr{R}(\langle\, E_1 \,\rangle (\langle\, E_2 \,\rangle W))$
$\langle\, E^* \,\rangle W$	$\mathscr{R}(W \vee \langle\, c(E) \,\rangle q)[q \leftarrow r_k]$
	where q is a fresh variable and $k = \lvert\mathscr{R}(W \vee \langle\, c(E) \,\rangle q)\rvert$
$\langle\, E \,\rangle X$	$\mathscr{R}(X) \wedge (\mathscr{R}(\langle\, E \,\rangle q) \uparrow k)[q \leftarrow r_k]$
	where q is a fresh variable and $k = \lvert\mathscr{R}(X)\rvert$
$\neg X$	$\mathscr{R}(X) \wedge (r_{k+1} \equiv \neg r_k)$
	where $k = \lvert\mathscr{R}(X)\rvert$
$X_1 \vee X_2$	$\mathscr{R}(X_1) \wedge \mathscr{R}(X_2) \uparrow j \wedge r_{j+k+1} \equiv (r_j \vee r_{k+j})$
	where $j = \lvert\mathscr{R}(X_1)\rvert$ and $k = \lvert\mathscr{R}(X_2)\rvert$

Table 3 FL$_r$ reduction rules of [51]

Example 10 Consider the right fusion logic formula of Example 8:

$$\langle\, \mathsf{step}(A)^* \,\rangle (B \vee C) \vee \langle\, \mathsf{step}(A) ; \mathsf{test}(B) \,\rangle D$$

This formula is reduced to $r_6 \wedge \Box_r I$, where

$$\begin{aligned} I = \ & (r_1 \equiv B \vee A) \wedge (r_2 \equiv A \wedge \bigcirc r_3) \wedge (r_3 \equiv r_1 \vee r_2) \wedge \\ & (r_4 \equiv D \wedge B) \wedge (r_5 \equiv A \wedge \bigcirc r_4) \wedge (r_6 \equiv r_3 \vee r_5). \end{aligned}$$

The number of dependent variables is 6, whereas with the new reduction technique we only get 2 dependent variables.

The new reduction technique is better, as both the size and number of dependent variables in the reduced formula is less than the one produced by the reduction technique of [51] because it only introduces a dependent variable for the 'step' and the 'chopstar' operator.

Reduction technique of Fisher et al. [21]: The reduction technique of Fisher et al. [21] reduces a Propositional Linear Time Temporal (PLTL) formula into Separated Normal Form (SNF) formula. Formulae in SNF are implications with present time formula on the left hand side and (present or) future-time formulae on the right-hand side. The general form of formula in SNF is $\Box \bigwedge_i A_i$ where each A_i is known as a PLTL clause and must be one of the following forms with each particular k_a, k_b, l_c, l_d, and l representing a literal (a propositional variable or its negation).

$$\begin{array}{rcll} \textbf{start} & \supset & \bigvee\limits_c l_c & \text{(initial PLTL clause)} \\ \bigwedge\limits_a k_a & \supset & \bigcirc \bigvee\limits_d l_d & \text{(step PLTL clause)} \\ \bigwedge\limits_b k_b & \supset & \Diamond l & \text{(sometime PLTL clause)} \end{array}$$

Let A be a PLTL formula and let $\mathbf{v}, \mathbf{y}$ and $\mathbf{z}$ be new propositional variables (indicated in bold face type). These variables serve the same role as our dependent variables. The following reduction rule introduces the start clause

$$\tau_0(A) = \Box(\textbf{start} \supset \mathbf{y}) \wedge \tau_1(\Box(\mathbf{y} \supset A))$$

A	$\tau_1(A)$
$\Box(x \supset (A \wedge B))$	$\tau_1(\Box(x \supset A)) \wedge \tau_1(\Box(x \supset B))$
$\Box(x \supset (A \supset B))$	$\tau_1(\Box(x \supset (\neg A \vee B))$
$\Box(x \supset (\neg(A \wedge B))$	$\tau_1(\Box(x \supset (\neg A \vee \neg B)))$
$\Box(x \supset \neg(A \supset B))$	$\tau_1(\Box(x \supset A)) \wedge \tau_1(\Box(x \supset \neg B))$
$\Box(x \supset \neg(A \vee B))$	$\tau_1(\Box(x \supset \neg A)) \wedge \tau_1(\Box(x \supset \neg B))$
$\Box(x \supset \bigcirc A)$	$\Box(x \supset \bigcirc \mathbf{y}) \wedge \tau_1(\Box(\mathbf{y} \supset A))$
	A neither literal nor disjunction of literals
$\Box(x \supset \neg \bigcirc A)$	$\Box(x \supset \bigcirc \mathbf{y}) \wedge \tau_1(\Box(\mathbf{y} \supset \neg A))$
$\Box(x \supset \Box A)$	$\tau_1(\Box(x \supset \Box \mathbf{y})) \wedge \tau_1(\Box(\mathbf{y} \supset A))$
	A not a literal
$\Box(x \supset \neg \Box A)$	$\Box(x \supset \Diamond \mathbf{y}) \wedge \tau_1(\Box(\mathbf{y} \supset \neg A))$
$\Box(x \supset \Diamond A)$	$\Box(x \supset \Diamond \mathbf{y}) \wedge \tau_1(\Box(\mathbf{y} \supset A))$
	A not a literal
$\Box(x \supset \neg \Diamond A)$	$\tau_1(\Box(x \supset \Box \mathbf{y})) \wedge \tau_1(\Box(\mathbf{y} \supset \neg A))$
$\Box(x \supset \Box l)$	$\tau_1(\Box(x \supset l)) \wedge \tau_1(\Box(x \supset \mathbf{y})) \wedge \Box(\mathbf{y} \supset \bigcirc l) \wedge \Box(\mathbf{y} \supset \bigcirc \mathbf{y})$
	l a literal
$\Box(x \supset D \vee A)$	$\tau_1(\Box(x \supset D \vee \mathbf{y})) \wedge \tau_1(\Box(\mathbf{y} \supset A))$
	D a disjunction of formula and
	A is neither a literal nor a disjunction of literals
$\Box(x \supset D)$	$\Box(\mathbf{start} \supset \neg x \vee D) \wedge \Box(\mathsf{true} \supset \bigcirc(\neg x \vee D))$
	D a literal or disjunction of literals
$\Box(x \supset \mathsf{true})$	$\Box(\mathbf{start} \supset \mathsf{true}) \wedge \Box(\mathsf{true} \supset \bigcirc \mathsf{true})$
$\Box(x \supset \mathsf{false})$	$\Box(\mathbf{start} \supset \neg x) \wedge \Box(\mathsf{true} \supset \bigcirc \neg x)$
$\Box(x \supset \Diamond l)$	$\Box(x \supset \Diamond l)$
	l a literal
$\Box(x \supset \bigcirc(l_1 \vee \ldots \vee l_n))$	$\Box(x \supset \bigcirc(l_1 \vee \ldots \vee l_n))$
	l_i a literal

Table 4 PLTL Reduction rules of [21]

The τ_1 reduction rules are listed in Table 4. We have omitted the rules for the PLTL temporal operators U (until) and W (weak until) in order to compare PLTL and FL. See [21] for these rules.

In the following example we reduce a formula using all three reduction techniques.

Example 11 In [21] the following PLTL formula $(\Diamond p \wedge \Box(p \supset \bigcirc p)) \supset \Diamond\Box p$ is reduced, yielding the formula in SNF $\Box \bigwedge_i A_i$, where the A_i's are listed below:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathbf{start} & \supset & \mathbf{f} & \mathbf{start} & \supset & (\neg\mathbf{q} \vee \neg p \vee \mathbf{s}) & \mathbf{start} & \supset & \neg\mathbf{f} \vee \mathbf{r} \\
\mathbf{f} & \supset & \Diamond p & \mathsf{true} & \supset & \bigcirc(\neg\mathbf{q} \vee \neg p \vee \mathbf{s}) & \mathsf{true} & \supset & \bigcirc(\neg\mathbf{f} \vee \mathbf{t}) \\
\mathbf{r} & \supset & \bigcirc\mathbf{q} & \mathbf{t} & \supset & \Diamond\neg p & \mathsf{true} & \supset & \bigcirc(\neg\mathbf{f} \vee \mathbf{r}) \\
\mathbf{r} & \supset & \bigcirc\mathbf{r} & \mathbf{u} & \supset & \bigcirc\mathbf{t} & \mathbf{start} & \supset & \neg\mathbf{f} \vee \mathbf{u} \\
\mathbf{start} & \supset & \neg\mathbf{f} \vee q & \mathbf{u} & \supset & \bigcirc\mathbf{u} & \mathbf{s} & \supset & \bigcirc p \\
\mathsf{true} & \supset & \bigcirc(\neg\mathbf{f} \vee \mathbf{q}) & \mathbf{start} & \supset & \neg\mathbf{f} \vee \mathbf{t} & \mathsf{true} & \supset & \bigcirc(\neg\mathbf{f} \vee \mathbf{u})
\end{array}$$

The PLTL formula is equivalent to the right FL formula

$$(\Diamond_r p \wedge \Box_r(p \supset \langle \mathsf{step}(\mathsf{true}) \rangle p)) \supset \Diamond_r\Box_r p.$$

Using the reduction rules of [51] on this right FL formula yields the formula $r_{24} \wedge \Box_r(\bigwedge_{i=1}^{24} r_i \equiv t_i)$, where the list of $r_i \equiv t_i$ formulae is as follows:

$$\begin{array}{rlrlrl}
r_1 & \equiv p & r_2 & \equiv \bigcirc r_3 & r_3 & \equiv r_1 \vee r_2 \\
r_4 & \equiv \neg r_3 & r_5 & \equiv \neg p & r_6 & \equiv \bigcirc p \\
r_7 & \equiv r_5 \vee r_6 & r_8 & \equiv \neg r_7 & r_9 & \equiv r_8 \\
r_{10} & \equiv \bigcirc r_{11} & r_{11} & \equiv r_9 \vee r_{10} & r_{12} & \equiv \neg r_{11} \\
r_{13} & \equiv \neg r_{12} & r_{14} & \equiv r_4 \vee r_{13} & r_{15} & \equiv \neg r_{14} \\
r_{16} & \equiv \neg r_{15} & r_{17} & \equiv \neg p & r_{18} & \equiv \bigcirc r_{19} \\
r_{19} & \equiv r_{17} \vee r_{18} & r_{20} & \equiv \neg r_{19} & r_{21} & \equiv r_{20} \\
r_{22} & \equiv \bigcirc r_{23} & r_{23} & \equiv r_{21} \vee r_{22} & r_{24} & \equiv r_{16} \vee r_{23}
\end{array}$$

Using the reduction rules introduced in this paper yields the formula
$(\neg\neg(\neg r_1 \vee \neg\neg r_2) \vee r_3) \wedge \bigwedge_{i=1}^{4} r_i \equiv t_i$, where the list of $r_i \equiv t_i$ formulae is as follows:

$$\begin{array}{rl}
r_1 & \equiv p \vee \bigcirc r_1 \\
r_2 & \equiv \neg(\neg p \vee \bigcirc p) \vee \bigcirc r_2 \\
r_3 & \equiv \neg r_4 \vee \bigcirc r_3 \\
r_4 & \equiv \neg p \vee \bigcirc r_4
\end{array}$$

The example shows that, for this example formula, the reduction technique introduced in this paper yields less dependent variables and clauses than the other two reduction techniques.

4.1.4 BDD Step

The BDD step adapts methods for symbolic state space traversal described by Coudert, Berthet and Madre [15–17] (see also [14, 40]) for use with BDD-based representations [6, 7] of formulas in propositional logic. It simultaneously greatly benefits from closely related methods first employed by McMillan in symbolic model checking [9, 14, 47] which also include the automatic generation of counterexamples for unsatisfiable formulas and, similarly, witnesses for satisfiable ones.

In this step, we transform the result of the reduction step, $init \wedge \Box_r \bigwedge_{i=1}^{k}(r_{X_i} \equiv t_i)$, into an automaton that is encoded using BDDs. We will now introduce BDDs Γ_i's used in the BDD encoding of $init \wedge \Box_r I$.

- Γ_1 represents the state formula *init*.
- Γ_2 captures all pairs of states corresponding to state intervals of length 1 (two states) satisfying invariant I. This can be done by replacing all variables in the scope of any $\bigcirc$ by a primed version and deleting the $\bigcirc$.
- Γ_3 captures the behavior of invariant I in an interval with only 1 state (the last state). This can be done by replacing each $\bigcirc$ construct by false, i.e., $\neg\mathsf{true}$.

The following example gives the Γ_i's for the formula of Example 8.

Example 12
In Example 8, we reduced $\langle\, \mathsf{step}(A)^* \,\rangle (B \vee C) \vee \langle\, \mathsf{step}(A)\,; \mathsf{test}(B) \,\rangle D$ to $\Box_r I \wedge init$, where *init* is equal to $r_{X_1} \vee r_{X_3}$ and I is equal to $(r_{X_1} \equiv (B \vee C) \vee (A \wedge \bigcirc r_{X_1})) \wedge (r_{X_3} \equiv A \wedge \bigcirc(B \wedge D))$.

- So Γ_1 is equal to $r_{X_1} \vee r_{X_3}$. The BDD encoding of $r_{X_1} \vee r_{X_3}$ is illustrated in Figure 4, where solid the oval nodes denote variables, and the box containing 1 represents the evaluation to true of the Boolean function. The solid lines denotes the assignment to true of a variable whereas the dashed line an assignment to false. The dotted line denotes the complement. The box with label 'initial state' is the name of the graph.

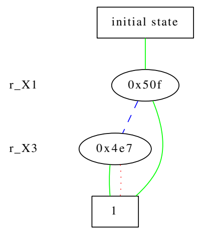


Fig. 4 BDD of initial state

- Γ_2 is obtained by replacing $\bigcirc var$ by var', i.e., Γ_2 is equal to $(r_{X_1} \equiv (B \vee C) \vee (A \wedge r'_{X_1})) \wedge (r_{X_3} \equiv A \wedge (B' \wedge D'))$.
  The BDD encoding of this formula is illustrated in Figure 5.

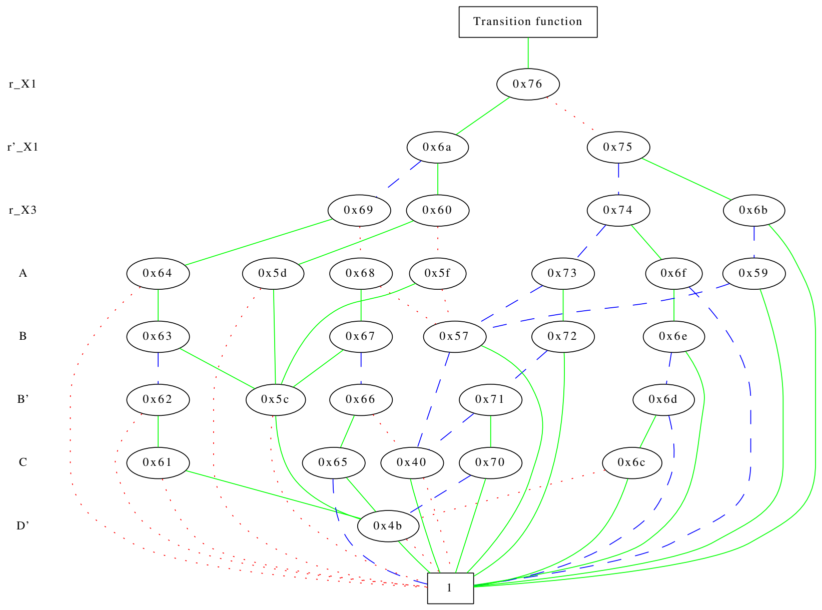


Fig. 5 BDD of transition relation

- Γ_3 is obtained by replacing $\bigcirc var$ by false, i.e., Γ_3 is equal to $(r_{X_1} \equiv (B \vee C) \vee (A \wedge \mathsf{false})) \wedge (r_{X_3} \equiv A \wedge \mathsf{false})$.

We now proceed to encode the satisfiability of a Fusion Logic formula into the satisfiability of a collection of BDDs.

- We use Γ_2 and Γ_1 to iteratively calculate a sequence of BDDs $\Delta_0, \ldots, \Delta_n$ so that for any n, Δ_n described all states which can be reached from Γ_1 in exactly n steps using Γ_2.
- We determine at each iteration whether BDD $\Gamma_3 \wedge \Delta_n$ is true or not. If false, we must continue to iterate; if true, then there exists some state satisfying Γ_3 which can be reached in n steps from Γ_1 and thus we can stop the iteration, i.e., the formula is satisfiable.

- During the iteration process, we maintain a BDD $\bigvee_{0\le i\le n} \Delta_i$ representing the set of all states so far reachable from Γ_1. If $(\bigvee_{0\le i\le n} \Delta_i) \equiv (\bigvee_{0\le i\le n+1} \Delta_i)$, i.e., no new states are found, then we stop the iteration. If we can not find a state that satisfies Γ_3, then the formula is not satisfiable.

To construct a satisfying interval for a satisfiable formula, we proceed as follows. Let Δ_m be that set of states for which $\Gamma_3 \wedge \Delta_m$ is true.

- If there are no independent variables (only r_i variables), then any interval of length m will satisfy the formula.
- If there are independent variables, then
  Find a value assignment σ_m for the independent variables for BDD Δ_m, i.e., choose one state σ_m of Δ_m.
  Compute Pr_{m-1} denoting those states of Δ_{m-1} that lead via Γ_2 to state σ_m (weakest precondition of Γ_2 and σ_m). Again choose one state σ_{m-1} of Pr_{m-1}.
  Continue until we reach Pr_0 and then choose state σ_0.
  The states $\sigma_0 \dots \sigma_{m-1}\sigma_m$ will then represent a (minimal) satisfying interval σ for the formula.

Implementation of the decision procedure: We have implemented the decision procedure for left Fusion Logic in the tool FLCheck[2] using the CUDD [64] BDD library. FLCheck is written in the scripting language Tcl/Tk which is a very convenient language for manipulating strings (in our case fusion logic formulae). The tool will check the validity or satisfiability of a FL_l formula. If a formula is not valid, the tool will produce a counter example, and if a formula is satisfiable, it will produce an example model.

4.2 Enforcement

When enforcing history-dependent policy rules, the main consideration is how to efficiently encode the policy in such a way that it can be evaluated against the input trace. By input trace, we refer to the sequence of variable assignments upon which the policy depends. The input trace will be used to fire policy rules, i.e., to determine whether the precondition of rules holds or not. The objective of the enforcer is, given an input trace, generate the corresponding access control decisions.

The enforcement of a policy P with respect to an input trace *Trace* corresponds to checking the satisfiability of $P \wedge \mathit{Trace}$. Let Trace_i $(0 \le i \le |\mathit{Trace}|)$ denote the i-th element in the input trace. As elements are added to *Trace*, $P \wedge \mathit{Trace}$ needs to be checked for satisfiability every time an element is added, i.e., we need to check satisfiability of the following formulae

$$\begin{array}{l} P \wedge \mathsf{empty}_l \left\langle \mathsf{test}(\mathit{Trace}_0) \right\rangle \\ P \wedge \mathsf{empty}_l \left\langle \mathsf{test}(\mathit{Trace}_0)\,;\mathsf{skip}\,;\mathsf{test}(\mathit{Trace}_1) \right\rangle \\ \dots \\ P \wedge \mathsf{empty}_l \left\langle \mathsf{test}(\mathit{Trace}_0)\,;\mathsf{skip}\dots;\mathsf{test}(\mathit{Trace}_{|\mathit{Trace}|}) \right\rangle \end{array}$$

We adapt our decision procedure to allow for enforcement with respect to input traces whereby we reuse previous results and thereby reduce the overhead of calculating the BDDs corresponding to P and the input trace. The enforcement procedure is as follows.

- As policy P is a left Fusion logic formula, we first rewrite it into a right Fusion logic formula P^r using the time reversal rules in Table 1.

[2] `http://www.tech.dmu.ac.uk/~cau/software/flcheck-1.0.tar.gz`

- We reduce P^r into the form $init \wedge \Box_r I$ using the reduction rules of Table 2.
- Generate the BDDs Γ_1 (initial state), Γ_2 (transition relation), and Γ_3 (final state) from *init* and invariant I as described in Section 4.1.4.
- As P is a $\boxed{\mathrm{i}}_l$ type of formula with semantics ‘for all prefix intervals’, the corresponding P^r is a $\Box_r$ formula, i.e., for all suffix intervals. The first state of P corresponds to the last state of P^r. So we know that the last state of P^r should satisfy both Γ_1 and Γ_3. Furthermore the last state should satisfy the first element of the input trace *Trace*. So the last state σ_0 of P^r (and the first state of P) is characterized by the BDD

$$\Gamma_1 \wedge \Gamma_3 \wedge Trace_0$$

- We then proceed as follows: we derive all the states that reach σ_0 via transition Γ_2 in one step, i.e., the weakest precondition denoted of Γ_2 and σ_0. Of those states, we take the one that satisfy Γ_1 and the second element in *Trace*, i.e., the BDD

$$Trace_1 \wedge \Gamma_1 \wedge \text{ weakest precondition}(\Gamma_2, \sigma_0)$$

  This BDD represents σ_1, the penultimate state of P^r (and the second state of P).
- We iteratively generate all the states corresponding to the elements in the input *Trace*, i.e.,

$$\sigma_{i+1} = (\Gamma_1 \wedge Trace_{i+1} \wedge \text{ weakest precondition}(\Gamma_2, \sigma_i))$$

- The sequence $\sigma_0, \ldots, \sigma_{|Trace|}$ corresponds to the sequence of states that should be generated while enforcing P with respect to input trace *Trace*, i.e., it represents the sequence of access control decisions.

Again, we have implemented above enforcement mechanism in our tool FLCheck. In the next section, we demonstrate the verification and enforcement of policies using various case studies.

5 Examples of Verification and Enforcement of Policies

In this section, we give examples of the verification of properties on access control policies using Fusion Logic in FLCheck. Furthermore, we show the enforcement of access control policies using FLCheck. All examples discussed in this section are included in the tool distribution.

5.1 Verification

Consider a system governed by role-based access control policies. The following sets are defined:

U	set of users $\{ac, hj\}$
R	set of roles $\{user, admin\}$
S	set of subjects $U \cup R$
O	set of objects $\{r, s\}$
A_r	set of actions $\{act_a, act_u, deact_a, deact_u\}$ on role manager object r
A_s	set of actions $\{create, access\}$ on server object s
A	set of actions $A_r \cup A_s$

– The user *ac* is assigned the role *admin*: $\mathrm{Aut}^d(ac,r,act_a), \mathrm{Aut}^d(ac,r,deact_a)$. Role assignment means that the subject (*ac*) can activate the role *and* deactivate the role at its discretion. We model removing the role assignment by denying the role activation (not taking into account that the role may already be activated).
– The user *hj* is assigned the role *user*: $\mathrm{Aut}^d(hj,r,act_u), \mathrm{Aut}^d(hj,r,deact_u)$.
– The role *admin* is allowed to *create* on server *s*: $\mathrm{Aut}^+(admin,s,create)$.
– The role *user* is allowed to *access* the server *s*: $\mathrm{Aut}^+(user,s,access)$.

Role Assignment: The user *ac* is unconditionally assigned the role *admin* and the user *hj* is unconditionally assigned the role *user*:

$$\begin{array}{l}\mathsf{true} \mapsto \mathrm{Aut}^+(ac,r,act_a)\\ \mathsf{true} \mapsto \mathrm{Aut}^+(ac,r,deact_a)\\ \mathsf{true} \mapsto \mathrm{Aut}^+(hj,r,act_u)\\ \mathsf{true} \mapsto \mathrm{Aut}^+(hj,r,deact_u)\end{array}$$

Consider in addition the conditional role-assignments stating that any user that has called in sick cannot (de-)activate any role with immediate effect. And that the user *hj* is temporarily promoted to act as administrator if *ac* is ill with immediate effect:

$$\begin{array}{l}0 :_l ill(u) \mapsto \mathrm{Aut}^-(u,r,A_r)\\ 0 :_l ill(ac) \mapsto \mathrm{Aut}^+(hj,r,act_a)\\ 0 :_l ill(ac) \mapsto \mathrm{Aut}^+(hj,r,deact_a)\end{array}$$

We assume here that the predicate $ill(u), u \in U$ can be directly observed. This is a slight simplification of the model, as otherwise we would have to capture the action of calling in sick on a managed object in the premise of the rule. Whilst this can be easily done, it would complicate the example unnecessarily. The prefix 0 : denotes that the predicate is to be evaluated in the same state (0 states in the past with respect to the state) in which the policy decision is made.
Permission Assignment: We assign unconditionally the permission $\langle s, create\rangle$ to the role *admin* and the permission $\langle s, access\rangle$ to the role *user*:

$$\begin{array}{l}\mathsf{true} \mapsto \mathrm{Aut}^+(admin,s,create)\\ \mathsf{true} \mapsto \mathrm{Aut}^+(user,s,access)\end{array}$$

Conflict Resolution, Decision Rule: As we define our RBAC example as a hybrid policy, e.g., both positive and negative authorizations are present in the same policy, we can create conflicts. It is not generally necessary to remove conflicts between positive and negative rules; however, there must be an unambiguous definition of which decision is being taken in case a conflict arises. We capture this in a standard decision rule (denial takes precedence):

$$\begin{array}{l}0 :_l (\mathrm{Aut}^+(U,r,A_r) \wedge \neg\mathrm{Aut}^-(U,r,A_r)) \mapsto \mathrm{Aut}^d(U,r,A_r)\\ 0 :_l (\mathrm{Aut}^+(R,s,A_s) \wedge \neg\mathrm{Aut}^-(R,s,A_s)) \mapsto \mathrm{Aut}^d(R,s,A_s)\end{array}$$

For the analysis of the policy, we expand the policy into its normal form by expanding the sets and then complete the policy specification with a set of default rules of the form $\mathsf{false} \mapsto c(s,o,a)$, where $c \in \{\mathrm{Aut}^d, \mathrm{Aut}^-, \mathrm{Aut}^+\}, s \in S, o \in O, a \in A_o$ and $a(s,o,a)$ does not occur as any consequence. It has been shown in [33] that the resulting policy is a refinement of the original specification. We refer to this specification as the model *M* of our policy.
Determining Conflicts: To check whether our policy model contains conflicting rules, we check the validity of $P \mathrel{\widehat{=}} \Box_l \neg(\mathrm{Aut}^+(s,o,a) \wedge \mathrm{Aut}^-(s,o,a))$, where $s \in S, o \in O, a \in A_o$.

The result of our decision procedure is that $M \supset P$ is not valid. A counter example is generated, for which the predicate $ill(ac)$ is true, leading to both the positive authorization $\mathrm{Aut}^{+}(ac, r, act_a)$ and the negative authorization $\mathrm{Aut}^{-}(ac, r, act_a)$. It remains to be checked whether for this conflict our decision rule yields the desired outcome. In this case, we are satisfied that the denial indeed takes precedence, and that *ac* is not allowed to act in the role *admin*.

Checking Dynamic Separation of Duty: We check whether the user *hj* and the user *ac* can possibly act in the role *admin* at the same time, i.e.,
$P \mathrel{\widehat{=}} \Box_l \neg(\mathrm{Aut}^d(ac, r, act_a) \wedge \mathrm{Aut}^d(hj, r, act_a))$. Querying our decision procedure yields the validity of $M \supset P$, so indeed *hj* and *ac* cannot possibly assume the role *admin* at the same time.

Checking Healthiness condition: The intent of conditionally promoting user *hj* was to ensure that there is always a user that can act in the role *admin*. This is a healthiness condition on the policy which can be checked by the property
$P \mathrel{\widehat{=}} \Box_l(\mathrm{Aut}^d(ac, r, act_a) \vee \mathrm{Aut}^d(hj, r, act_a))$. Checking the validity of $M \supset P$ yields the somewhat unexpected result that it is not valid. The counter example generated provides that if *both* users are ill, i.e., $ill(ac) \wedge ill(hj)$ holds, no user can activate the role *admin*.

Checks with additional Assumptions: Given the above result, one may decide to ignore the case that all users are ill, based on the observation that there would be nobody to use the system. To ascertain whether this is the only aspect of the policy that resulted in the failure of the previous validity check, we provide the additional assumption $A \mathrel{\widehat{=}} \Box_l \neg(ill(ac) \wedge ill(hj))$. Rechecking the validity of the healthiness condition under the assumption A, i.e., checking $(M \wedge A) \supset P$ for validity results in true. This means that A is a sufficient condition to ensure the healthiness condition.

The static checking of policies for consistency and other properties, such as dynamic separation of duty constraints or domain dependent healthiness conditions, is important before the policy is deployed to the policy decision points (PDP). However, in the PDPs the emphasis is on the efficient evaluation of policies.

5.2 Enforcement

We now check for the enforcement of a policy against a defined input trace using our FLCheck tool.

Input Trace:	w_0	w_1	w_2	w_3
$ill(ac)$	0	1	1	0
$ill(hj)$	0	0	1	1
$\mathrm{Aut}^d(ac, r, act_a)$	1	0	0	1
$\mathrm{Aut}^d(ac, r, act_u)$	0	0	0	0
$\mathrm{Aut}^d(hj, r, act_a)$	0	1	0	0
$\mathrm{Aut}^d(hj, r, act_u)$	1	1	0	0
$\mathrm{Aut}^-(ac, r, act_a)$	0	1	1	0
$\mathrm{Aut}^-(ac, r, act_u)$	0	1	1	0
$\mathrm{Aut}^-(hj, r, act_a)$	0	0	1	1
$\mathrm{Aut}^-(hj, r, act_u)$	0	0	1	1
$\mathrm{Aut}^+(ac, r, act_a)$	1	1	1	1
$\mathrm{Aut}^+(ac, r, act_u)$	0	0	0	0
$\mathrm{Aut}^+(hj, r, act_a)$	0	1	1	0
$\mathrm{Aut}^+(hj, r, act_u)$	1	1	1	1

We see that initially both users *ac* and *hj* are not ill. Consequently, only *ac* was allowed to activate the role admin. Then, *ac* fell ill and *hj* is able to act as administrator. In the third step, w_2 states that both users are ill, and therefore no role activation is permitted. In the last state, *ac* returns to work and can act as administrator.

Our approach allows for the incremental enforcement with respect to an input trace, i.e., it uses pre-computed BDDs (Γ_1, Γ_2, and Γ_3). The number of nodes used in the BDDs is independent of the length of the input trace. If we had used the decision procedure without modification, the number of nodes would directly depend on the length of the input trace.

5.3 Enforcement with History

Let us now consider an example where the history of execution is decisive to the access control decision.

In a multi-modal authentication/authorization scheme, a `user` has to present two tokens (KA and KB) to the access control mechanism in order to gain `access` to a `resource`. The tokens must be presented in a certain order:

- if KB is currently present, then KA should have been present $2 * n$ states ($n \geq 1$) before the current state. So we need:

$$\begin{array}{cccccl}
 & & \bullet & \bullet & \bullet & \text{or} \\
 & & \text{KA} & & \text{KB} & \\
\hline
\bullet & \bullet & \bullet & \bullet & \bullet & \text{or} \\
\text{KA} & & & & \text{KB} & \\
\hline
 & & & & \ldots &
\end{array}$$

- if KA is currently present, then KB should have been present $1 + 2 * n$ states ($n \geq 0$) before the current state. So we need:

$$\begin{array}{ccccl}
 & & \bullet & \bullet & \text{or} \\
 & & \text{KB} & \text{KA} & \\
\hline
\bullet & \bullet & \bullet & \bullet & \text{or} \\
\text{KB} & & & \text{KA} & \\
\hline
 & & & \ldots &
\end{array}$$

For simplicity, we do not consider positive and negative authorizations here and model the policy as a closed policy, e.g., access is denied unless explicitly stated otherwise. The access control policy rule is expressible in left fusion logic as follows:

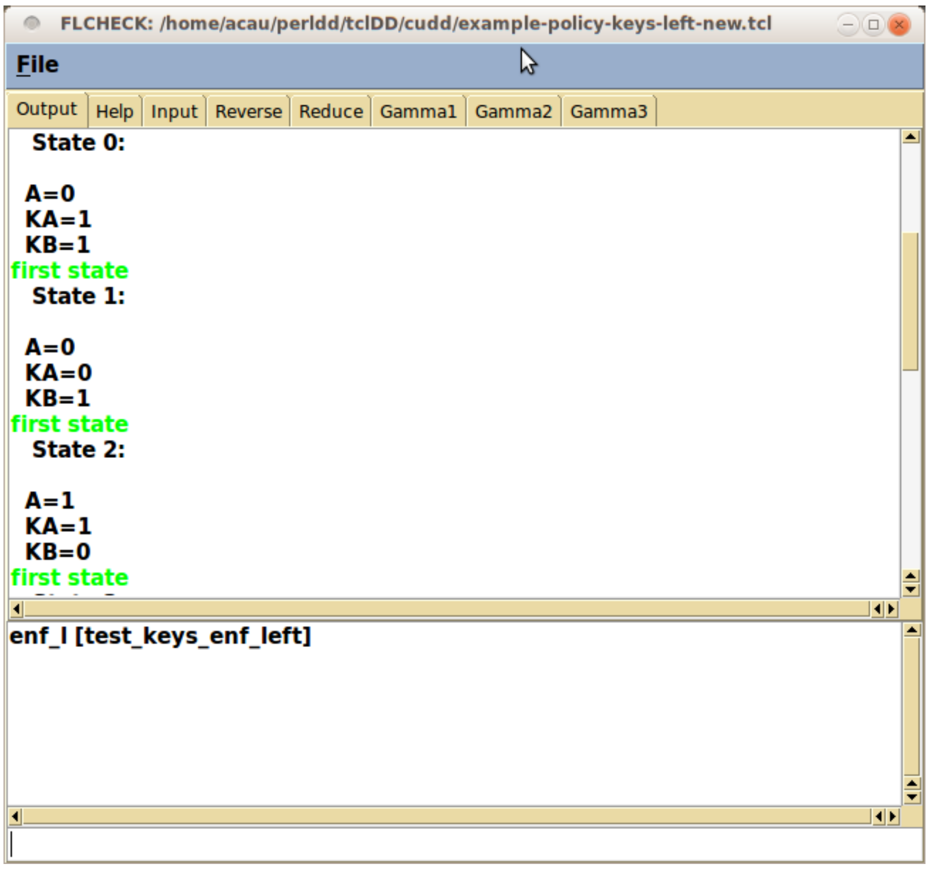
$$\begin{array}{r}
\mathsf{true} \left\langle \mathsf{test}(\mathrm{KA}) \,;\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}) \,;(\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}))^* \,;\mathsf{test}(\mathrm{KB}) \right\rangle \vee \\
\mathsf{true} \left\langle \mathsf{test}(\mathrm{KB}) \,;\mathsf{step}(\mathsf{true}) \,;(\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}))^* \,;\mathsf{test}(\mathrm{KA}) \right\rangle
\end{array} \mapsto A$$

where $A \mathrel{\widehat{=}} \mathrm{Aut}^d(\mathit{user}, \mathit{resource}, \mathit{access})$.

Because a complete specification is needed for enforcement, we proceed as before and strengthen the specification by explicitly specifying the conditions under which the access decision is false, i.e., $\mathit{Pre} \leftrightarrow W = \mathit{Pre} \mapsto W \wedge \neg \mathit{Pre} \mapsto \neg W$. Therefore we also have

$$\begin{array}{r}
\neg(\mathsf{true} \left\langle \mathsf{test}(\mathrm{KA}) \,;\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}) \,;(\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}))^* \,;\mathsf{test}(\mathrm{KB}) \right\rangle \vee \\
\mathsf{true} \left\langle \mathsf{test}(\mathrm{KB}) \,;\mathsf{step}(\mathsf{true}) \,;(\mathsf{step}(\mathsf{true}) \,;\mathsf{step}(\mathsf{true}))^* \,;\mathsf{test}(\mathrm{KA}) \right\rangle)
\end{array} \mapsto \neg A$$

The table below shows the values for KA and KB that where used in one example enforcement run, together with the corresponding access control decision A:


FLCHECK: /home/acau/perldd/tclDD/cudd/example-policy-keys-left-new.tcl
File
Output
Help
Input
Reverse
Reduce
Gamma1
Gamma2
Gamma3

```
 State 0:

A=0
KA=1
KB=1
first state
 State 1:

A=0
KA=0
KB=1
first state
 State 2:

A=1
KA=1
KB=0
first state
```

```
enf_l [test_keys_enf_left]
```

Fig. 6 Enforcement Run

Input Trace:	w_0	w_1	w_2	w_3	w_4	w_5	w_6
A	0	0	1	1	1	0	1
KA	1	0	1	1	0	0	1
KB	1	1	0	0	1	0	1

In state w_0, access is not granted because KB is currently present, but KA was not present an even (not zero) number of states before and KA is currently present, but KB was not present an odd number of states before. In state w_1, access is not granted because KB is currently present, but KA was not present an even (not zero) number of states before. In state w_2, access is allowed because KA is currently present and KB was present 1 state before. In state w_3, access is allowed because KA is currently present and KB was present 3 states before. In state w_4, the access is allowed because KB is currently present and KA was present 2 (and 4) states before. In state w_5, access is not allowed because both KA and KB are not currently present. In state w_6, access is allowed because KA is currently present and KB was present 2 (and 6) states before and KB is currently present and KA was present 3 states before.

Figure 6 shows a screen shot of FLCheck during this run.

6 Conclusion and Future Work

In this paper we have shown how a formal policy model can be expressed in left Fusion Logic. We have then shown how the formulae, representing our access control policies, are reduced to a normal form that can be encoded using Binary Decision Diagrams (BDD). Using our tool FLCheck, we analyzed a variety of properties of a simple RBAC policy, where we focused on the role assignment. These properties included consistency of the policy, traditional safety checks and domain dependent healthiness conditions. We have shown how domain-dependent assumptions can be integrated in the check without modifying the model (i.e., policy) itself. We then used the inductive nature of our decision procedure to check against an externally provided input trace, *reusing* the previously generated BDDs.

In our future work, we plan to encode first-order logic constructs to provide a more flexible history-based policy language as used in [33].

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments that were very helpful for shaping the manuscript.

References

1. Abadi M, Fournet C (2003) Access control based on execution history. In: 10th Annual Network and Distributed System Symposium (NDSS'03)., The Internet Society, Reston, Virginia, USA, pp 1–15
2. Antimirov VM (1996) Partial derivatives of regular expressions and finite automaton constructions. Theoretical Compututer Science 155(2):291–319, DOI 10.1016/0304-3975(95)00182-4
3. Bahrak B, Deshpande A, Whitaker M, Park JM (2010) Bresap: A policy reasoner for processing spectrum access policies represented by binary decision diagrams. In: New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on, pp 1 –12, DOI 10.1109/DYSPAN.2010.5457867
4. Bandara AK, Lupu EC, Sloman M (2007) Policy-Based Management. In: Burgess M, Bergstra J (eds) Handbook of Network and System Administration, Elsevier, chap Policy Based Management
5. Bertino E, Bonatti PA, Ferrari E (2001) TRBAC: A temporal role-based access control model. ACM Trans Inf Syst Secur 4(3):191–233, DOI 10.1145/501978.501979
6. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691, DOI 10.1109/TC.1986.1676819
7. Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3):293–318, DOI 10.1145/136035.136043
8. Brzozowski JA (1964) Derivatives of regular expressions. J ACM 11:481–494, DOI 10.1145/321239.321249
9. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 1020 states and beyond. Inf Comput 98(2):142–170, DOI 10.1016/0890-5401(92)90017-A
10. Chaochen Z, Hoare CAR, Ravn AP (1991) A calculus of durations. Inf Process Lett 40(5):269–276, DOI 10.1016/0020-0190(91)90122-X

11. Church A (1957) Applications of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic – Volume 1, Cornell Univ., pp 3–50
12. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma E, Larsen KG (eds) CAV, Springer, Lecture Notes in Computer Science, vol 2404, pp 359–364
13. Cimatti A, Roveri M, Tonetta S (2008) Symbolic compilation of PSL. IEEE Trans on CAD of Integrated Circuits and Systems 27(10):1737–1750, DOI 10.1109/TCAD.2008.2003303
14. Clarke EM Jr, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge, MA, USA
15. Coudert O, Berthet C, Madre JC (1989) Verification of sequential machines using boolean functional vectors. In: Claesen L (ed) Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, pp 111–128
16. Coudert O, Berthet C, Madre JC (1989) Verification of synchronous sequential machines based on symbolic execution. In: Sifakis J (ed) Automatic Verification Methods for Finite State Systems, International Workshop, Springer, Grenoble, France, no. 407 in LNCS, pp 365–373
17. Coudert O, Berthet C, Madre JC (1990) A unified framework for the formal verification of sequential circuits. In: Proc. IEEE International Conf. on Computer Aided Design, pp 126–129
18. Damianou N, Dulay N, Lupu E, Sloman M (2001) The Ponder specification language. In: Workshop on Policies for Distributed Systems and Networks (Policy2001)
19. Duan Z, Tian C, Zhang L (2008) A decision procedure for propositional projection temporal logic with infinite models. Acta Inf 45:43–78, DOI 10.1007/s00236-007-0062-z
20. Elgaard J, Klarlund N, Møller A (1998) MONA 1.x: New techniques for WS1S and WS2S. In: Proc. 10th International Conference on Computer-Aided Verification, CAV '98, Springer-Verlag, LNCS, vol 1427, pp 516–520
21. Fisher M, Dixon C, Peim M (2001) Clausal temporal resolution. ACM Transactions on Computational Logic 2(1):12–56, DOI 10.1145/371282.371311
22. Gomez R, Bowman H (2004) PITL2MONA: Implementing a decision procedure for propositional interval temporal logic. Journal of Applied Non-Classical Logics 14(1-2):105–148, issue on Interval Temporal Logics and Duration Calculi. V. Goranko and A. Montanari guest eds.
23. Harel D (1987) Statecharts: A visual formalism for complex systems. Sci Comput Program 8(3):231–274, DOI 10.1016/0167-6423(87)90035-9
24. Harel D, Kozen D, Tiuryn J (2000) Dynamic Logic. MIT Press, Cambridge, MA
25. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Software Eng 23(5):279–295
26. Hu H, Ahn GJ, Kulkarni K (2011) Anomaly discovery and resolution in web access control policies. In: Proceedings of the 16th ACM symposium on Access control models and technologies, ACM, New York, NY, USA, SACMAT '11, pp 165–174, DOI 10.1145/1998441.1998472
27. IBM Cooperation (2003) Enterprise Privacy Authorisation Language (EPAL) Version 1.2. submitted to the W3C, URL `http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/`
28. IEEE (2005) IEEE Standard for Property Specification Language (PSL), std 1850-2005. Tech. rep., IEEE, DOI 10.1109/IEEESTD.2005.97780

29. ISO/IEC (2006) ISO/IEC 10181-3:1996 Information technology – Open Systems Interconnection – Security frameworks for open systems: Access control framework. URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=18199`
30. Jajodia S, Samarati P, Sapino ML, Subrahmanian VS (2001) Flexible support for multiple access control policies. ACM Trans Database Syst 26(2):214–260, DOI 10.1145/383891.383894
31. Janicke H, Cau A, Siewe F, Zedan H, Jones K (2006) A compositional event & time-based policy model. In: Proceedings of POLICY2006, London, Ontario, Canada, IEEE Computer Society, London, Ontario Canada, pp 173–182
32. Janicke H, Cau A, Siewe F, Zedan H (2007) Deriving enforcement mechanisms from policies. In: Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007), pp 161–170
33. Janicke H, Cau A, Siewe F, Zedan H (2012) Dynamic access control policies: Specification and verification. The Computer Journal DOI 10.1093/comjnl/bxs102
34. Joshi J, Bertino E, Ghafoor A (2005) An analysis of expressiveness and design issues for the generalized temporal role-based access control model. IEEE Trans Dependable Sec Comput 2(2):157–175, DOI 10.1109/TDSC.2005.18
35. Joshi J, Bertino E, Latif U, Ghafoor A (2005) A generalized temporal role-based access control model. IEEE Trans Knowl Data Eng 17(1):4–23, DOI 10.1109/TKDE.2005.1
36. Klarlund N, Møller A (2001) MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University, notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3
37. Kolovski V (2008) Logic-based framework for web access control policies. PhD thesis, Computer Science Department of University of Maryland, College Park
38. Kono S (1995) A combination of clausal and non-clausal temporal logic programs. In: Fisher M, Owens R (eds) Executable Modal and Temporal Logics, Springer Verlag, Cambery, France, Lecture Notes in Artificial Intelligence, vol 897, pp 40–57
39. Kozen D (1992) private commuication
40. Kropf T (1999) Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems, 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA
41. Lamport L (1994) The temporal logic of actions. ACM Trans Program Lang Syst 16(3):872–923, DOI 10.1145/177492.177726
42. Lampson BW (1974) Protection. SIGOPS Oper Syst Rev 8(1):18–24, DOI 10.1145/775265.775268
43. Leucker M, Sánchez C (2010) Regular linear-time temporal logic. In: Markey N, Wijsen J (eds) TIME, IEEE Computer Society, pp 3–5, DOI 10.1109/TIME.2010.29
44. Lupu EC, Sloman M (1999) Conflicts in policy-based distributed systems management. IEEE Trans Softw Eng 25(6):852–869, DOI 10.1109/32.824414
45. Manna Z, Pnueli A (1992) The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York
46. Masood A, Ghafoor A, Mathur AP (2010) Conformance testing of temporal role-based access control systems. IEEE Trans Dependable Sec Comput 7(2):144–158, DOI 10.1109/TDSC.2008.41
47. McMillan KL (1993) Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA
48. Moszkowski B (1983) Reasoning about digital circuits. PhD thesis, Department of Computer Science, Stanford University, technical report STAN–CS–83–970

49. Moszkowski B (1985) A temporal logic for multilevel reasoning about hardware. IEEE Computer 18(2):10–19
50. Moszkowski B (2004) A hierarchical completeness proof for Propositional Interval Temporal Logic with finite time. Journal of Applied Non-Classical Logics 14(1–2):55–104, special issue on Interval Temporal Logics and Duration Calculi
51. Moszkowski B (2005) A hierarchical analysis of propositional temporal logic based on intervals. In: Artemov S, Barringer H, d'Avila Garcez AS, Lamb LC, Woods J (eds) We Will Show Them: Essays in Honour of Dov Gabbay, vol 2, College Publications (formerly KCL Publications), King's College, London, pp 371–440
52. Moszkowski B (2011) Compositional reasoning using intervals and time reversal. Temporal Representation and Reasoning, International Syposium on 0:107–114, DOI 10.1109/TIME.2011.25
53. OASIS (2005) eXtensible Access Control Markup Language (XACML) Version 2.0. URL `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20`
54. Pandya PK (2001) Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. In: Pattersson P, Yovine S (eds) Proceedings of Workshop on Real-time Tools (RT-TOOL'2001), Affiliated with CONCUR 2001, Uppsala University, Sweden, vol Technical Report 2001-14, full version available as Technical Report TCS-00-PKP-1, Tata Institute of Fundamental Research, 2000
55. Park J, Sandhu RS (2004) The UCON$_{ABC}$ usage control model. ACM Trans Inf Syst Secur 7(1):128–174, DOI 10.1145/984334.984339
56. Park J, Zhang X, Sandhu RS (2004) Attribute mutability in usage control. In: Farkas C, Samarati P (eds) Proceedings of IFIP TC11/WG 11.3 Eighteenth Annual Conference on Data and Applications Security, Kluwer, Sitges, Catalonia, Spain, pp 15–29
57. Prior AN (1967) Past, Present and Future. Oxford University Press
58. Rao P, Lin D, Bertino E, Li N, Lobo J (2009) An algebra for fine-grained integration of xacml policies. In: Proceedings of the 14th ACM symposium on Access control models and technologies, ACM, New York, NY, USA, SACMAT '09, pp 63–72, DOI 10.1145/1542207.1542218
59. Sandhu R, Park J (2003) The UCON$_{ABC}$ usage control model. In: Proceeding of the Second International Workshop on Mathematical Method, Models and Architectures for Computer Networks Security
60. Schaad A, Lotz V, Sohr K (2006) A model-checking approach to analysing organisational controls in a loan origination process. In: Ferraiolo DF, Ray I (eds) SACMAT, ACM, pp 139–149, DOI 10.1145/1133058.1133079
61. Schneider FB (2000) Enforceable security policies. ACM Transactions on Information and System Security 3(1):30–50, DOI 10.1145/353323.353382
62. Siewe F, Cau A, Zedan H (2003) A compositional framework for access control policies enforcement. In: proceedings of the ACM workshop on Formal Methods in Security Engineering: From Specifications to Code
63. Sloman M (1994) Policy driven management for distributed systems. Journal of Network and Systems Management 2:333–360
64. Somenzi F (1998) CUDD: Colorado University Decision Diagram package. University of Colorado at Boulder, URL `http://vlsi.colorado.edu/~fabio/CUDD/`
65. Thomas W (2008) Church's problem and a tour through automata theory. In: Avron A, Dershowitz N, Rabinovich A (eds) Pillars of computer science, Springer-Verlag, Berlin, Heidelberg, pp 635–655

66. Woo TYC, Lam SS (1993) Authorization in distributed systems: A new approach. Journal of Computer Security 2(2,3):107–136
67. Zhang X, Park J, Parisi-Presicce F, Sandhu R (2004) A logical specification for usage control. In: SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies, ACM Press, New York, NY, USA, pp 1–10, DOI 10.1145/990036.990038
68. Zhang X, Parisi-Presicce F, Sandhu RS, Park J (2005) Formal model and policy specification of usage control. ACM Trans Inf Syst Secur 8(4):351–387, DOI 10.1145/1108906.1108908
69. Zhang X, Sandhu RS, Parisi-Presicce F (2006) Safety analysis of usage control authorization models. In: Lin FC, Lee DT, Lin BS, Shieh S, Jajodia S (eds) ASIACCS, ACM, pp 243–254, DOI 10.1145/1128817.1128853
70. Zhang X, Seifert JP, Sandhu RS (2008) Security enforcement model for distributed usage control. In: Singhal M, Serugendo GDM, Tsai JJP, Lee WC, Römer K, Tseng YC, Hsiao HCW (eds) SUTC, IEEE Computer Society, pp 10–18, DOI 10.1109/SUTC.2008.79

A Proofs

Lemma 3

Let R be a right fusion logic formula and $dep(R)$ be the dependent variables introduced by $\mathscr{R}'_k(R)$ and $\mathscr{R}_k(R)$ $(k = 0, 1)$. Then the formula below is valid:

$$\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$$

Proof The proof is done via structural induction on R and by employing the reduction rules in Table 2. We also make use of some existing Fusion Logic theorems.

- Case $R = W$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(W)$ and $dep(W) = \emptyset$, $\Box_r \mathsf{true}$ which is equal to true.
- Case $R = \neg X$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(\neg X)$, $\exists dep(\neg X) \bullet \Box_r \mathscr{R}_k(X)$. Because $dep(\neg X) = dep(X)$, this is equal to $\exists dep(X) \bullet \Box_r \mathscr{R}_k(X)$. Using the induction hypothesis, this holds.
- Case $R = X_1 \vee X_2$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(X1 \vee X_2)$, $\exists dep(X_1 \vee X_2) \bullet \Box_r(\mathscr{R}_k(X_1) \wedge \mathscr{R}_k(X_2))$ is equal to, as $dep(X_1 \vee X_2) = dep(X_1) \cup dep(X_2)$ and $dep(X_1) \cap dep(X_2) = \emptyset$, $\exists dep(X_1) \bullet \Box_r \mathscr{R}_k(X_1) \wedge \exists dep(X_2) \bullet \Box_r \mathscr{R}_k(X_2)$. Using the induction hypothesis twice, this holds.
- Case $R = \langle\, \mathsf{test}(W)\, \rangle X$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(\langle\, \mathsf{test}(W)\, \rangle X)$, $\exists dep(\langle\, \mathsf{test}(W)\, \rangle X) \bullet \Box_r \mathscr{R}_k(X)$. This is equal to, as $dep(\langle\, \mathsf{test}(W)\, \rangle X) = dep(X)$, $\exists dep(X) \bullet \Box_r \mathscr{R}_k(X)$. Using the induction hypothesis, this holds.
- Case $R = \langle\, \mathsf{step}(T)\, \rangle X$: We first consider $k = 0$. $\exists dep(R) \bullet (\Box_r \mathscr{R}_0(R))$ is equal to, using the reduction rule $\mathscr{R}_0(\langle\, \mathsf{step}(T)\, \rangle X)$, $\exists dep(\langle\, \mathsf{step}(T)\, \rangle X) \bullet \Box_r(r_{\langle\, \mathsf{step}(T)\, \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X)) \wedge \mathscr{R}_0(X))$. This is equal to, as $dep(\langle\, \mathsf{test}(W)\, \rangle X) = \{r_{\langle\, \mathsf{step}(T)\, \rangle X}\} \cup dep(X)$, $\exists r_{\langle\, \mathsf{step}(T)\, \rangle X} \bullet (\Box_r(r_{\langle\, \mathsf{step}(T)\, \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X)))) \wedge \exists dep(X) \bullet \Box_r \mathscr{R}_0(X)$. This is equal to, using the induction hypothesis, $\exists r_{\langle\, \mathsf{step}(T)\, \rangle X} \bullet (\Box_r(r_{\langle\, \mathsf{step}(T)\, \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X))))$. Substitute for $r_{\langle\, \mathsf{step}(T)\, \rangle X}$ the value $(T \wedge \bigcirc \mathscr{R}'_0(X))$ will give $\Box_r((T \wedge \bigcirc \mathscr{R}'_0(X)) \equiv (T \wedge \bigcirc \mathscr{R}'_0(X)))$ which holds.
  We now consider the case $k = 1$. $\exists dep(R) \bullet \Box \mathscr{R}_1(R)$ is equal to, using the reduction rule $\mathscr{R}_1(\langle\, \mathsf{step}(T)\, \rangle X)$, $\exists dep(\langle\, \mathsf{step}(T)\, \rangle X) \bullet \Box_r \mathscr{R}_0(X)$. This is equal to, as $dep(\langle\, \mathsf{test}(W)\, \rangle X) = dep(X)$, $\exists dep(X) \bullet \Box_r \mathscr{R}_0(X)$. Using the induction hypothesis, this holds.
- Case $R = \langle\, E_1 \vee E_2\, \rangle X$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(\langle\, E_1 \vee E_2\, \rangle X)$ and $dep(\langle\, E_1 \vee E_2\, \rangle X) = dep(\langle\, E_1\, \rangle X) \cup dep(\langle\, E_2\, \rangle X)$ and using the reduction rule $\mathscr{R}_k(\langle\, E_1\, \rangle X \vee \langle\, E_2\, \rangle X)$, $\exists dep(\langle\, E_1\, \rangle X) \cup dep(\langle\, E_2\, \rangle X) \bullet \Box_r(\mathscr{R}_k(\langle\, E_1\, \rangle X) \wedge \mathscr{R}_k(\langle\, E_2\, \rangle X))$. Using the induction hypothesis in a similar way as the case $R = X_1 \vee X_2$, we get the desired result.

- Case $R = \langle E_1 ; E_2 \rangle X$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(\langle E_1 ; E_2 \rangle X)$, let $X_1 = \langle E_2 \rangle X$ and $dep(\langle E_1 ; E_2 \rangle X) = dep(\langle E_1 \rangle X_1)$, $\exists dep(\langle E_1 \rangle X_1) \bullet \Box_r \mathscr{R}_k(\langle E_1 \rangle X_1)$. Using the induction hypothesis, this holds.
- Case $R = \langle E^* \rangle X$: $\exists dep(R) \bullet \Box_r \mathscr{R}_k(R)$ is equal to, using the reduction rule $\mathscr{R}_k(\langle E^* \rangle X)$ and let X_1 denote $X \vee \langle c(E) \rangle r_{\langle E^* \rangle X}$, $\exists dep(\langle E^* \rangle X) \bullet (\Box_r(r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1)) \wedge \Box_r \mathscr{R}_1(X_1))$. Using induction hypothesis and $dep(\langle E^* \rangle X) = \{r_{\langle E^* \rangle X}\} \cup dep(X)$, we get $\exists r_{\langle E^* \rangle X} \bullet \Box_r(r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1))$. Substitute for $r_{\langle E^* \rangle X}$ the value $\mathscr{R}'_1(X_1)$, will give the desired result.

QED

Lemma 4

Let R be a right fusion logic formula. Then the next implication is valid:

$$\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$$

Proof The proof is done via structural induction on R and by employing the reduction rules in Table 2. We also make use of some existing Fusion Logic theorems.

- Case $R = W$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(W)$ and $\mathscr{R}_k(W)$, $\Box_r\mathsf{true} \supset W \equiv W$ and this holds.
- Case $R = \neg X$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rule $\mathscr{R}'_k(\neg X)$ and $\mathscr{R}_k(\neg X)$, $\Box_r \mathscr{R}_k(X) \supset ((\neg \mathscr{R}'_k(X)) \equiv \neg X)$ which is equivalent to $\Box_r \mathscr{R}_k(X) \supset (\mathscr{R}'_k(X) \equiv X)$. Use the induction hypothesis to get the desired result.
- Case $R = X_1 \vee X_2$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(X_1 \vee X_2)$ and $\mathscr{R}_k(X1 \vee X_2)$, $\Box_r(\mathscr{R}_k(X_1) \wedge \mathscr{R}_k(X_2)) \supset ((\mathscr{R}'_k(X_1) \vee \mathscr{R}'_k(X_2)) \equiv (X_1 \vee X_2))$. Using the induction hypothesis $\Box_r \mathscr{R}_k(X_1) \supset (\mathscr{R}'_k(X_1) \equiv X_1)$ and $\Box_r \mathscr{R}_k(X_2) \supset (\mathscr{R}'_k(X_2) \equiv X_2)$, and the theorem $\Box_r(\mathscr{R}_k(X_1) \wedge \mathscr{R}_k(X_2)) \equiv \Box_r(\mathscr{R}_k(X_1)) \wedge \Box_r(\mathscr{R}_k(X_2))$, will give the desired result.
- Case $R = \langle \mathsf{test}(W) \rangle X$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(\langle \mathsf{test}(W) \rangle X)$ and $\mathscr{R}_k(\langle \mathsf{test}(W) \rangle X)$, $\Box_r \mathscr{R}_k(X) \supset ((W \wedge \mathscr{R}'_k(X)) \equiv \langle \mathsf{test}(W) \rangle X)$. Using the induction hypothesis $\Box_r \mathscr{R}_k(X) \supset (\mathscr{R}'_k(X) \equiv X)$ and the theorem $\langle \mathsf{test}(W) \rangle X \equiv (W \wedge X)$, gives the desired result.
- Case $R = \langle \mathsf{step}(T) \rangle X$: We first consider $k = 0$. $\Box_r \mathscr{R}_0(R) \supset (\mathscr{R}'_0(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_0(\langle \mathsf{step}(T) \rangle X)$ and $\mathscr{R}_0(\langle \mathsf{step}(T) \rangle X)$, $\Box(r_{\langle \mathsf{step}(T) \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X)) \wedge \mathscr{R}_0(X)) \supset (r_{\langle \mathsf{step}(T) \rangle X} \equiv \langle \mathsf{step}(T) \rangle X)$. We know that $\Box_r(r_{\langle \mathsf{step}(T) \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X)) \wedge \mathscr{R}_0(X))$ is equivalent to $\Box_r(r_{\langle \mathsf{step}(T) \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X))) \wedge \Box_r \mathscr{R}_0(X)$. Using the induction hypothesis $\Box_r \mathscr{R}_0(X) \supset (\mathscr{R}'_0(X) \equiv X)$ we get $(\Box(r_{\langle \mathsf{step}(T) \rangle X} \equiv (T \wedge \bigcirc \mathscr{R}'_0(X))) \wedge \Box_r \mathscr{R}_0(X)) \supset (\mathscr{R}'_0(X) \equiv X)$. Using the theorem $(T \wedge \bigcirc \mathscr{R}'_0(X)) \equiv \langle \mathsf{step}(T) \rangle \mathscr{R}'_0(X)$, gives us the desired result.
  We now consider the case $k = 1$. $\Box_r \mathscr{R}_1(R) \supset (\mathscr{R}'_1(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_1(\langle \mathsf{step}(T) \rangle X)$ and $\mathscr{R}_1(\langle \mathsf{step}(T) \rangle X)$, $\Box_r \mathscr{R}_0(X) \supset ((T \wedge \bigcirc \mathscr{R}'_0(X)) \equiv \langle \mathsf{step}(T) \rangle X)$. Using the induction hypothesis $\Box_r \mathscr{R}_0(X) \supset (\mathscr{R}'_0(X) \equiv X)$ and the theorem $(T \wedge \bigcirc \mathscr{R}'_0(X)) \equiv \langle \mathsf{step}(T) \rangle \mathscr{R}'_0(X)$, gives us the desired result.
- Case $R = \langle E_1 \vee E_2 \rangle X$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(\langle E_1 \vee E_2 \rangle X)$, $\mathscr{R}'_k(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$, $\mathscr{R}_k(\langle E_1 \vee E_2 \rangle X)$ and $\mathscr{R}_k(\langle E_1 \rangle X \vee \langle E_2 \rangle X)$, $\Box(\mathscr{R}_k(\langle E_1 \rangle X) \wedge \mathscr{R}_k(\langle E_2 \rangle X)) \supset ((\mathscr{R}'_k(\langle E_1 \rangle X) \vee \mathscr{R}'_k(\langle E_2 \rangle X)) \equiv \langle E_1 \vee E_2 \rangle X)$. Using the induction hypothesis $\Box_r \mathscr{R}_k(\langle E_1 \rangle X) \supset (\mathscr{R}'_k(\langle E_1 \rangle X) \equiv \langle E_1 \rangle X)$ and $\Box_r \mathscr{R}_k(\langle E_2 \rangle X) \supset (\mathscr{R}'_k(\langle E_2 \rangle X) \equiv \langle E_2 \rangle X)$ plus the theorem $\Box_r(\mathscr{R}_k(\langle E_1 \rangle X) \wedge \mathscr{R}_k(\langle E_2 \rangle X)) \equiv (\Box_r \mathscr{R}_k(\langle E_1 \rangle X)) \wedge (\Box_r \mathscr{R}_k(\langle E_2 \rangle X))$ and $\langle E_1 \vee E_2 \rangle X \equiv \langle E_1 \rangle X \vee \langle E_2 \rangle X$, gives us the desired result.
- Case $R = \langle E_1 ; E_2 \rangle X$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(\langle E_1 ; E_2 \rangle X)$ and $\mathscr{R}_k(\langle E_1 ; E_2 \rangle X)$, $\Box_r \mathscr{R}_k(\langle E_1 \rangle \langle E_2 \rangle X) \supset (\mathscr{R}'_k(\langle E_1 \rangle \langle E_2 \rangle X) \equiv \langle E_1 ; E_2 \rangle X)$. Using the induction hypothesis $\Box_r \mathscr{R}_k(\langle E_1 \rangle X_1) \supset (\mathscr{R}'_k(\langle E_1 \rangle X_1) \equiv \langle E_1 \rangle X_1)$ and $\Box_r \mathscr{R}_k(\langle E_2 \rangle X) \supset (\mathscr{R}'_k(\langle E_2 \rangle X) \equiv \langle E_2 \rangle X)$ and the theorem $\langle E_1 ; E_2 \rangle X \equiv \langle E_1 \rangle \langle E_2 \rangle X$, gives us the desired result.
- Case $R = \langle E^* \rangle X$: $\Box_r \mathscr{R}_k(R) \supset (\mathscr{R}'_k(R) \equiv R)$ is equal to, using the reduction rules $\mathscr{R}'_k(\langle E^* \rangle X)$ and $\mathscr{R}_k(\langle E^* \rangle X)$ and let X_1 denote $X \vee \langle c(E) \rangle r_{\langle E^* \rangle X}$, $\Box_r((r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1)) \wedge \mathscr{R}_1(X_1)) \supset (r_{\langle E^* \rangle X} \equiv \langle E^* \rangle X)$. Using the induction hypothesis $\Box_r \mathscr{R}_1(X_1) \supset (\mathscr{R}'_1(X_1) \equiv X_1)$ and the theorem $\Box_r((r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1)) \wedge \mathscr{R}_1(X_1)) \equiv (\Box_r(r_{\langle E^* \rangle X} \equiv \mathscr{R}'_1(X_1))) \wedge (\Box_r \mathscr{R}_1(X_1))$ and $\langle E^* \rangle X \equiv X \vee \langle c(E) \rangle \langle E^* \rangle X$, gives us the desired result.

QED