Deriving Enforcement Mechanisms from Policies

Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan
Software Technology Research Laboratory,
De Montfort University, Leicester, UK LE1 9BH
Email: {heljanic, acau, fsiewe, hzedan}@dmu.ac.uk

***Abstract*— Policies provide a flexible and scalable approach to the management of distributed systems by separating the specification of security requirements and their enforcement. Over the years the expressiveness of policy languages increased considerably making it possible to capture a variety of complex requirements that for example depend on the history of the system execution. The most important criteria for the successful operation of policy-managed systems is whether the deployed enforcement mechanisms can *guarantee* the compliance with the policies. With the expressiveness of policy languages this assurance is increasingly difficult to achieve. In this paper we therefore address the development of enforcement mechanisms from a theoretical perspective and show how enforcement code can be formally derived for compositional, history-dependent policies that can change dynamically over time or on the occurrence of events.**

I. Introduction

Managing large-scale distributed systems in which resources are made available to a large number of subjects, potentially across organisational boundaries, is a challenging task. Ensuring the security, especially the control of access to resources, is paramount as the unauthorised use of resources, the disclosure or modification of information can have critical consequences to the business success of an organisation. A policy-based approach is advantageous in that many security requirements can be expressed at a high level of abstraction [1]. Policies are enforced by dedicated, possibly certified, components in the system. This modularity leads to systems that are easier to maintain and allows for the timely adoption of new requirements if the need arises. Timeliness is especially important as the time between identification of a vulnerability and the implementation of countermeasures is critical.

Research on policies has mainly concentrated on the development of more expressive languages and models for the specification of security and management requirements. With the increased expressiveness tool-support has been developed to assist in the specification process and also to support the analysis w.r.t. properties such as conflicts, completeness or information-flow. However, increased expressiveness affects not only the specification of policies, but also the design and implementation of concrete enforcement mechanisms. The gap between policy specification and the implementation of enforcement mechanisms can lead to insecure systems, as the overall security of the system relies heavily on the correctness of the enforcement mechanism. Typically support for the enforcement is provided together with the policy languages. These mechanisms are however mostly developed without an underlying formal model and rely solely on the developer to have matched the semantics of the policy language. To bridge the gap between policy specification and the development of concrete enforcement mechanisms we show in this paper how enforcement code that guarantees the correct enforcement of the policy can be derived.

A well known and widely used architecture for policy enforcement is the Reference Monitor (RM). The RM intercepts all access requests to the resources and accepts or rejects them based on the policy. A modular approach to RM implementations [2] separates the enforcement and the policy evaluation into a Policy Enforcement Point (PEP), viz. the component intercepting the request, and the Policy Decision Point (PDP), viz. the component that evaluates an access request against a policy and reports the decision back to the PEP. Many logic-based policy languages [3], [4] reduce the problem of enforcement to a decidability problem and use a variant of Datalog [5] to actually make policy decisions. This approach, whilst providing a concise definition of how policy decisions are made, addresses only part of the enforcement — namely the PDP. The correct deployment of PEPs and mechanisms to observe and manage system events that the policy evaluation depends on are typically not addressed.

A class of policy models, for which the observation of events is especially important, is history-based access control [6]. History-based access control allows to use more contextual information in the specification than other policy models, e.g. stack-based, activity-based. This increases the number of requirements that can be expressed, but also requires more sophisticated enforcement mechanisms. The key difficulties with the enforcement of this class of policies are:

- Determining the history required for policy decisions.
- Maintaining the history.
- Optimisation of enforcement efficiency.
- Timeliness of policy enforcement.

In this paper we address these points using a formal approach to the development of enforcement mechanisms. We base our discussion on our formal policy model [7]–[9] that can express history-based access control policies. The advantage of our model is that it allows for the concise, specification of history dependencies using Interval Temporal Logic (ITL) and additionally is compositional, viz. large complex policies can be composed out of smaller easier comprehensible policies along a structural and temporal axis. Composition along the temporal axis allows for the specification of policy changes

over time or on the occurrence of events.

By modelling access control requests within the same formal framework, we are able to specify and reason about the interaction of system behaviour, i.e. sequences of accesses made, and the mechanisms enforcing the policies. From the policies we derive (executable) enforcement code that corresponds to the behaviour of the enforcement mechanism. We do not propose to implement this code directly as "hard-wired" part of the distributed system, but envision the automation of the described process so that the enforcement code can be automatically *compiled* to a loosely coupled module within the RM, that can be easily exchanged if need arises. We believe that compilation has significant advantages in its potential to optimise for efficiency over the more traditionally used approaches where the PDP is interpreting the policy.

The structure of the remainder of the paper is as follows. In Sections II and III we introduce the underlying logic and key aspects of the policy model. We then formalise the behaviour of a RM in Section V and show how policies, defined at a higher level of abstraction, can be mapped to the low level RM implementation in Section VI. Having established the link between policy and RM we then transform policy rules and policy compositions into provably correct enforcement code as part of the RM's behaviour. The history dependencies of the policy are analysed to determine the amount of history information that is required for the enforcement of the policy. In Section VIII we review related work in this area and compare it with our approach. We conclude the paper in Section IX where we also outline future work.

II. Preliminary

We use Interval Temporal Logic (ITL) to define policy decisions over sequences of states (behaviours). The motivation to use ITL is that it provides a compositional way of defining policies that determine the outcome of policy decisions. Especially important for this work is the linkage of ITL specifications at different granularity of time via the *projection* operator. This is used to map between policies that are defined in terms of requests to the much finer grained implementation of the RM. We use the word implementation here, because many well-known programming constructs can be expressed directly in ITL, allowing for a straightforward translation of the RM into procedural programming languages.

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0,\ \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of integer values $\mathbb{Z}$. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

The syntax of ITL is defined in Figure 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

Expressions	
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
Formulae	
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$ $\mathsf{skip} \mid f_1 ; f_2 \mid f^*$

Fig. 1. Syntax of ITL

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- f_1 ; f_2: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Figure 2.

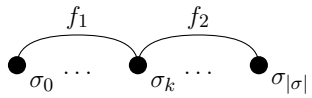

Fig. 2. Informal Semantics of f_1 ; f_2

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Figure 3.

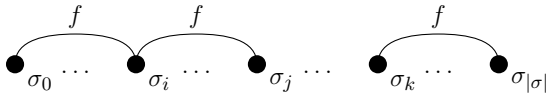

Fig. 3. Informal Semantics of f^*

- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- fin v: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

A. Derived Constructs

The following is a list of some derived constructs that are used in the remainder of this paper.

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} ; f$	next f, f holds from the next state. Example: $\bigcirc(X = 1)$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.
$\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$	non-empty interval, i.e., any interval of length at least one.
$\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$	interval, i.e., any interval of length zero (just one state).
$\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true} ; \mathsf{false}$	infinite interval, i.e., any interval of infinite length.

$\mathsf{finite} \mathrel{\widehat{=}} \neg\,\mathsf{inf}$ finite interval, i.e., any interval of finite length.

$\Diamond f \mathrel{\widehat{=}} \mathsf{finite}\,;f$ sometimes f, i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ always f, i.e., any interval such that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.

$\diamondsuit f \mathrel{\widehat{=}} f\,;\mathsf{true}$ diamond-i, i.e., any interval such that f holds over a prefix sub-interval.

$\boxdot f \mathrel{\widehat{=}} \neg \diamondsuit \neg f$ box-i, i.e., any interval such that f holds over all prefix sub-intervals.

$\circledast f \mathrel{\widehat{=}} \Diamond(\diamondsuit f)$ diamond-a, i.e., any interval such that f holds over a sub-interval.

$\boxplus f \mathrel{\widehat{=}} \neg \circledast(\neg f)$ box-a, i.e., any interval such that f holds over all sub-intervals.

$\mathsf{keep}\ f \mathrel{\widehat{=}} \boxplus(\mathsf{skip} \supset f)$ keep f, i.e., any interval such that f holds over all unit sub-intervals.

$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$ final state, i.e., any interval such that f holds in the final state of that interval.

$v := e \mathrel{\widehat{=}} (\bigcirc v) = e$ assignment, i.e., the value of v will be e in the next state.

$v \leftarrow e \mathrel{\widehat{=}} \mathsf{finite} \wedge (\mathsf{fin}\ v) = e$ temporal assignment, i.e., the value of v in the final state will be the value of e.

$\mathsf{stable}\ v \mathrel{\widehat{=}} \Box(\mathsf{more} \supset v := v)$ remain stable, i.e., the value of v remains stable in the interval.

B. Temporal Projection

Using an ITL concept called *temporal projection* [10], we can provide a high-level specification of enforcement mechanisms that defines an appropriate level of abstraction and links the policy with the system implementation. Temporal projection allows to relate specifications that are defined on a different granularity of time. The formula $f \Delta g$ holds over an interval σ iff *i)* the formula g is true on some interval σ' obtained by projecting some states from σ and *ii)* the formula f is true on each of the subintervals of σ bridging the gaps between the projected states. The semantics of $f \Delta g$ is:

$$\begin{aligned}\sigma \models\ f \Delta g \text{ iff} \quad & \text{for some } n \geq 0, \sigma' \text{ and } l_0, \ldots, l_n : \\ & 0 = l_0 < \cdots < l_n = |\sigma|, \text{ and} \\ & \text{for each } i < n,\ \sigma_{l_i} \ldots \sigma_{l_{i+1}} \models f, \\ & \text{and } \sigma' \models g \text{ where } |\sigma'| = n \text{ and} \\ & \text{for all } i \leq n,\ \sigma'_i = \sigma_{l_i}\end{aligned}$$

Here $\sigma \models f$ denotes that the interval σ models f. It can be shown that the operator f^* can be expressed in terms of the projection operator as $f \Delta \mathsf{true}$. More detail on the projection operator and its properties can be found in [10].

C. Memory Variables

For an ITL specification to be complete, the values of all state variables must be defined for every state. This can be cumbersome when using ITL for the specification of programs, where the general assumption is that a variable maintains its value, unless assigned differently. We refer to these variables as memory variables. A convenient notation for specifications using memory variables is to define the assignment with respect to a set of memory variables:

$$[\![v \leftarrow e]\!]_V \mathrel{\widehat{=}} v \leftarrow e \wedge \bigwedge_{x \in V \setminus \{v\}} \mathsf{stable}\ x$$

In this paper all variables used in the enforcement code are memory variables. We therefore relax the notation for presentation purposes and omit the enclosing $[\![]\!]_V$.

III. Policy Model

We focus our discussion on the enforcement of dynamically changing policies [7]–[9]. The model can express history and state-based access control and obligation requirements in a compositional manner, viz. the overall system policy can be composed out of smaller, more comprehensible policies along a structural and temporal axis. Composition along the temporal axis allows to capture the dynamic change of policies on time or the occurrence of events. This makes the model especially well-suited for the development of work-flow-based systems, where the execution is staged in phases.

Compositionality is an important aspect for the specification and verification of policies, as it leads to more modular specifications that can also be easier maintained. Additionally it improves the performance of enforcement mechanisms as these can take advantage of the additional information provided by the policy composition. In this section we provide an overview of our policy language and its semantics, as this is the starting point for the development of enforcement mechanisms. For more detailed descriptions of our policy language we refer the reader to e.g. [7]–[9].

A. Policy Rules

Policy rules are the basic building block of our policy model. Typically a rule captures single requirements such as "administrators can delete files". Rules are structured in a premise and a consequence. We distinguish between authorisation and obligation rules dependent on the rules' consequences. For the above example the consequence would for example be $\mathrm{autho}^+(s, \mathrm{file}, \mathrm{delete})$. The superscript $^+$ here indicates a *positive* authorisation; $^-$ indicates a negative authorisation. Since the specification of both can lead to conflicts so called *decision rules* are defined that do not carry a superscript. The use of these superscripts is limited to authorisation rules; obligations (oblig) do not carry superscripts.

Especially interesting for this work is the premise of a rule. It defines the condition under which the rule *fires*.

Our model defines the premise to be a negation free subset of ITL formulae (though negation in state formulae, viz. formulae without temporal operator, is permissible). This is not a major limitation, as the negation of formulae represents a set of behaviours that is not intuitive and constructive. Since our aim is to derive concrete enforcement code we require policy specifications to be constructive. However using the algebra of the underlying logic, many negated formulae can be transformed into their duals, moving the negation inside to satisfy above constraint.

The premise describes a set of behaviours. The intuition is that the observation of this behaviour in *some* past interval, as seen from the point in which the policy decision is made, leads to the consequence to be true. This is formally defined by the operator *always-followed-by* ($\mapsto$) that connects the premise and the consequence of a rule. For the above example the rule would be:

$$\mathrm{in}(s, \mathrm{admin}) \;\mapsto\; \mathrm{autho}^{+}(s, \mathrm{file}, \mathrm{delete})$$

The formal semantics of the operator is given in (1); its informal semantics is depicted in Figure 4.

$$f \;\mapsto\; w \mathrel{\widehat{=}}_{\boxdot} ((\Diamond f) \;\supset\; \mathsf{fin}\; w) \tag{1}$$

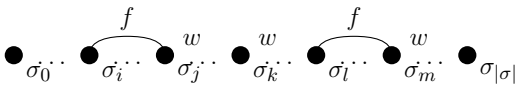

Fig. 4. Weak Always-Followed-By

We will focus in Section VII on the analysis of rule premises and their transformation into enforcement code. Another key aspect of the policy model is its compositionality.

B. Composed Policies

Under policy composition we understand the formal composition of two policies to form a larger policy to capture the overall requirement. The advantage of a compositional approach is that properties of the composition can be inferred from properties of its parts. This is important for the verification and validation of policies, but also provides an advantage for the enforcement as we discuss in Section VII-E.

Our compositional approach is especially well suited for policies that are applied in work-flow systems, where protection requirements are typically dependent on the current phase of the system. Our unique notion of temporal composition allows to easily express the change of policies when the system makes a transition from one phase to another. Due to the space limitations in this paper we cannot detail the semantics of the composition operators and only provide a simple example of a composition to serve as a representative for the approach:

$$\langle \mathrm{event} \rangle P \,;\, Q$$

This temporal composition informally means that: unless the event event is observed, the policy P is enforced, subsequently the policy Q. Formally the operator unless ($\langle w \rangle : P$) is defined as:

$$\langle w \rangle : P \mathrel{\widehat{=}} (((P \wedge \Box w)\,;\mathsf{skip}) \wedge \mathsf{fin}\, \neg w) \vee (\mathsf{empty} \wedge \neg w)$$

We refer interested readers to [7]–[9] for a more detailed discussion of other composition operators.

IV. Enforcement

Policies, as described in Section III, define policy decisions over sequences of states. The transition from one state to the next corresponds to the time between two policy decisions, viz. the processing of a single request. The *correct* enforcement of a policy means:

a) Authorisation: It is always the case that if an action a has been successfully performed by a subject s on an object o then the same subject has been authorised in the previous state to perform action a on object o.

b) Obligation: It is always the case that if the RM mediates an action a on an object o then the RM was under no obligation in the previous state.

We express this formally as the following enforcement properties E_{autho} and E_{oblig}, where the Boolean state variable $\mathrm{done}(s, o, a)$ denotes the successful execution of the action a on the object o by the subject s:

$$E_{\mathrm{autho}} \mathrel{\widehat{=}} \mathsf{keep}\,(\bigcirc \mathrm{done}(s, o, a) \supset\; \mathrm{autho}(s, o, a)) \tag{2}$$

$$E_{\mathrm{oblig}} \mathrel{\widehat{=}} \mathsf{keep}\,(\bigcirc \mathrm{done}(s, o, a) \supset\; \neg\, \mathrm{oblig}(\mathrm{RM}, o', a')) \tag{3}$$

Any correct RM implementation must *guarantee* that these enforcement properties are satisfied. To ensure this, the RM has to maintain the relevant history of the execution in between two consecutive policy states and place an appropriate condition on the execution of an action. As this will involve some form of computation it is clear that the behaviour of the RM is defined over a much finer-grained interval than the policy. We present in Section V the behavioural specification of the RM and show in Section VI how the policy and enforcement properties are mapped onto this behaviour.

V. Reference Monitor

Subjects can request the execution of an action on an object. This request is checked by the Reference Monitor (RM) and either granted or rejected. Figure 5 depicts this process.

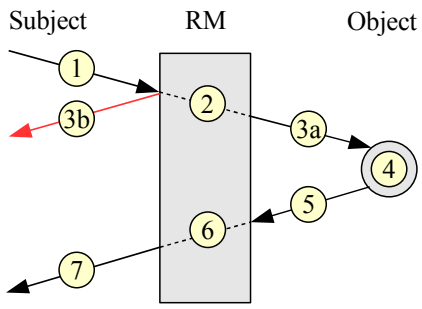

Fig. 5. Reference Monitor

1 Subject requests the execution of an action on an object. It passes the required parameters as part of the request.
2 The RM intercepts this request and evaluates the policy to determine whether to grant or reject the request[1], taking the passed parameters into account. The RM also maintains the relevant history of accesses needed for future policy decisions.
3a Provided that the request is granted, the request is passed to the object.
3b A rejected request is is indicated to the requesting subject.
4 The object executes the requested action and updates its state accordingly.
5 It sends the results, e.g. output parameter or confirmation messages back to the requesting subject.
6 The RM intercepts the response of the object and maintains the relevant history (e.g. results).
7 The RM passes the response to the requesting subject.

The RM mediates the access to a set of objects that are under its protection. This set is in the following denoted by $\mathcal{O}$. Alternatively to the mediation of a request, the RM can execute obligations that result from the enforced policy. We assume that the number of subjects requesting access to these objects is finite and that all subjects can be authenticated. The set of subjects is denoted by $\mathcal{S}$. The set of actions that can be executed on the objects is typically defined by the types of the objects themselves. We assume that for every object $o \in \mathcal{O}$ the set of corresponding actions $\mathcal{A}_o$ that can be executed is known.

A. Access Request

For every action $a \in \mathcal{A}_o$ which can be performed on an object $o \in \mathcal{O}$ a list of variables $V_{a,in}$ holding the actions input parameters and a list of variables $V_{a,out}$ holding the actions output parameters are given. We then define a single access request as:

$$\begin{aligned} request(s,o,a,V_{a,in},V_{a,out}) \mathrel{\widehat{=}} & \ enf_{pre}; \\ & \text{if } C_{\text{autho}} \text{ then } \ exec\ ;\ enf_{post}\ ;\ succeed; \\ & \text{else } fail \end{aligned} \tag{4}$$

A single request is divided into phases as explained in Figure 5. Here enf_{pre} represents the enforcement code that must be executed before the access. For example to to maintain the history that is relevant to determine whether a request is granted or rejected. C_{autho} is the concrete access control decision for the access. If the request is granted, the corresponding action is executed ($exec$) and enforcement code (enf_{post}) that must be executed after the action, e.g. to maintain a history of output parameters, is performed. If the access control decision resulted in a rejection of the request, the action is considered to be failed. We define $succeed$ and $fail$ as:

$$succeed \mathrel{\widehat{=}} \text{done}(s,o,a), \overline{\text{done}(s,o,a)} \leftarrow \mathsf{true}, \overline{\mathsf{false}} \tag{5}$$

$$fail \mathrel{\widehat{=}} \text{failed}(s,o,a), \overline{\text{failed}(s,o,a)} \leftarrow \mathsf{true}, \overline{\mathsf{false}} \tag{6}$$

[1]We do not distinguish here between the policy decision point (PDP) and the policy enforcement point (PEP). Using the ISO model [2], the PEP would make a request to the PDP to establish whether to grant or reject the request.

The Boolean state variables $\text{done}(s,o,a)$ and $\text{failed}(s,o,a)$ are used as control variables that indicate the success or failure of the request and can be used in policy specifications. We denote by $\overline{\text{done}(s,o,a)}$ the list of all control variables except $\text{done}(s,o,a)$. Similarly for $\overline{\text{failed}(s,o,a)}$ the respective lists of complementary control variables. By $\overline{\mathsf{false}}$ we denote a list of false literals of the same length. We do not detail the concrete execution in this paper, as it does not have a direct influence on future policy decisions and only assume that $exec$ is finite.

The RM mediates the access to the objects under its protection. We assume that the RM does only mediate one access request at a time. This currently presents a major limitation in our enforcement model which is being addressed in future work. The behaviour of the RM can then be described as in (7).

$$RMS \mathrel{\widehat{=}} \left(\begin{array}{l} \text{if } C_{\text{oblig}} \text{ then } exec_{\text{oblig}} \text{ else} \\ \left(\bigoplus_{s\in\mathcal{S},o\in\mathcal{O},a\in\mathcal{A}} request(s,o,a,V_{a,in},V_{a,out}) \oplus \mathsf{skip} \right) \end{array} \right)^* \tag{7}$$

Where $\oplus$ denotes the operator *exclusive-or*. If the condition of an obligation is met then the RM will execute the obligation (denoted by $exec_{\text{oblig}}$), otherwise it will process a request or remain idle. We do not detail the execution of the obligation, as it is not directly relevant to the objective of the paper. The aim of a *correct* RM implementation is to define the enforcement enf_{pre}, enf_{post}, and the conditions C_{autho} and C_{oblig} in such a way that the enforcement properties E_{autho}, and E_{oblig} are satisfied w.r.t. the enforced policy. This requires to adequately link the policy specification and the RM implementation.

VI. Linking System Implementation and Policy

The policy model described in Section III defines the behaviour of the variables $\text{autho}(s,o,a)$ and $\text{oblig}(s,o,a)$ over an interval. We denote in the following the intervals defined by the policy as σ'. Similarly, (7) defines the possible behaviours of the RM at the implementation level. We denote a RM interval in the following by σ. Both specifications are linked, as the policy may reference the state and the history of the execution using for example the variables $\text{done}(s,o,a)$, and the system execution depends on the authorisation and obligation decisions that are defined by the policy.

Policy specifications are more abstract than the concrete implementation of the system. The level of abstraction must be carefully chosen, as policies specified on too fine-grained intervals can make the implementation of correct enforcement mechanisms infeasible and policies specified on too coarse intervals may not provide the required level of protection. For example, if the policy is specified at the same granularity as the system implementation (meaning $\sigma = \sigma'$), then the specification of $1 : P$ (viz. the policy P is enforced for an interval of length 1) in the policy would take the same time as a variable assignment in the implementation. If a number of rules needs to be evaluated to make a policy decision then their evaluation at the implementation level may well be infeasible within this short time span, thus violating the

original specification. To ensure a suitable abstraction, we project the states in the intervals σ' onto the interval σ using *temporal projection.*

A. Marking Projected States

Given the behaviour of the RM in (7), we choose to project the policy specification on the initial state of enf_{pre}. The input parameters of the processed request are available, but no decision on the success or failure of the request has yet been made. Furthermore, it means that the transition from a state σ'_i to σ'_{i+1} in the policy specification covers the execution of *exactly* one request in the system implementation. Consequently all effects that result from the execution of a request are considered for the next policy decision.

To be able to project the policy specification onto this particular state, we need to mark it. For this purpose we introduce a marker state variable m that is *only* true in this state. Figure 6 depicts the behaviour of a single access request that corresponds to (4) with the values of the marker m.

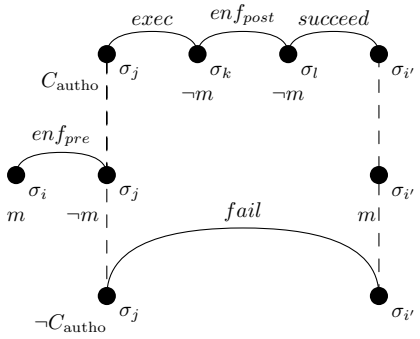

Fig. 6. Marker m for a single request.

After the pre-enforcement, the condition C_{autho} determines whether the access is executed (top sequence) or the request fails (bottom sequence). The marker is only true in states σ_i, viz the initial state of the pre-enforcement. We change the definition of a request accordingly:

$$
\begin{aligned}
&request(s, o, a, V_{a,in}, V_{a,out}) \mathrel{\widehat{=}} \\
&\quad m{=}\mathsf{true} \wedge enf_{pre}; \\
&\quad \text{if } C_{\text{autho}} \text{ then} \\
&\qquad exec \,;\, enf_{post} \,;\, succeed \\
&\quad \text{else } fail
\end{aligned}
\tag{8}
$$

It can be shown that the introduction of the marker m in (8) is a refinement (subset relationship on sets of intervals) of the original specification (4). The details are available in [7].

B. Projection

We now use the marker m for the projection. Let rm be the specification of the reference monitor implementation:

$$
\begin{aligned}
rm &\mathrel{\widehat{=}} RMS \wedge (M \Delta (P \wedge E_{\text{autho}} \wedge E_{\text{oblig}})) \\
M &\mathrel{\widehat{=}} (m \wedge \mathsf{skip}) \,;\, ((\mathsf{keep}\ \neg m) \wedge \mathsf{fin}\,(m))
\end{aligned}
\tag{9}
$$

Where M bridges the projected states (initially $m = \mathsf{true}$ followed by an interval in which $m = \mathsf{false}$ in all but the last state). P is the policy specification on the projected states. The mapping between the policy specification and the implementation of the RM is depicted in Figure 7.

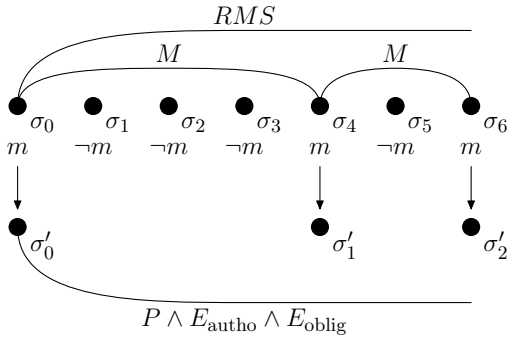

Fig. 7. Mapping between Policy and RM implementation

In Figure 7 the interval σ that represents the behaviour of the RM at the implementation level is depicted together with the values of the marker state variable m. States σ_i for which m is true are projected. The policy and the enforcement specification are defined over the interval σ' consisting of the projected states. Deriving concrete enforcement mechanisms from the policy specification means to define enf_{pre}, enf_{post}, C_{autho} and C_{oblig} of each request in such a way, that P is correctly enforced, i.e. that RMS satisfies the enforcement properties E_{autho} and E_{oblig}.

VII. Deriving Enforcement Code

The policy model described in Section III is expressive, as it allows to define authorisation and obligation constraints, based on the observed behaviour of the system. We assume in the following, that the policy specification uses only information and events that can be accessed by the RM. For example if we consider Multi-Level Security (MLS), then the RM must be able to obtain the security labels of the objects and the clearance levels of the subjects in order to enforce the policy.

We show how the RM can maintain the history of this information to ensure that the policy is still enforced correctly. The approach minimises the amount of history information that must be kept, and is therefore suitable for the enforcement of history-dependent policies in domains where *i)* a large number of requests are being made, and *ii)* the time that is required to make a decision is a critical factor. In the following we will concentrate the discussion on authorisation rules — the enforcement of obligation rules is similar.

A. Enforcing State-Dependent Rules

Policy rules are of the form: $premise \mapsto consequence$. Where the premise is an ITL formula defining a set of behaviours that when observed in the past lead to the consequence in the current state.

A state-dependent rule has the form $0 : w \mapsto \text{autho}(s, o, a)$. The rule does not refer to any past state or

events — its enforcement is consequently straightforward. At any projected state σ'_i where w is true, s is authorised to perform a on o. In the implementation of the enforcement code enf_{pre} and enf_{post} we ensure that none of the variables that w can depend on is modified, viz. the enforcement does not have side effects. This ensures that the evaluation of w in σ'_i yields the same value as if evaluated in the final state of enf_{pre} of the current request, in which the condition C_{autho} is evaluated.

By defining the condition C_{autho} of the corresponding request to be w, we ensure that the request fails if w is false. The failure of the request (see (6)) guarantees that the corresponding control variables $\text{failed}(s, o, a)$ is set to true and $\text{done}(s, o, a)$ is set to false. As we assume that all variables are stable unless explicitly assigned it follows that in the next projected state $\text{done}(s, o, a)$ will still be false. This clearly satisfies the enforcement property E_{autho} which can also be expressed as $\neg\,\text{autho}(s, o, a) \supset \bigcirc\neg\,\text{done}(s, o, a)$.

B. Enforcing Temporal-dependent Rules

Rules with a temporal dependency have the form $t : f \mapsto \text{autho}(s, o, a)$, where $0 < t \leq T$. T is the time since the enforcement of the rule started. Alternatively the rule can omit the explicit timing which can be shown to be equivalent to $T : \Diamond f \mapsto \text{autho}(s, o, a)$, viz. we only need to consider the first case. Given the explicit timing of a rule, we can establish the maximal history that is required for its enforcement.

For example the rule $1 : \neg\,\text{in}(s, \text{admin}) \mapsto \text{autho}(s, o, a)$ states that a subject s can perform a on o if it was not acting in the role admin in the state before. For simplicity we assume here that $\text{in}(s, r)$ is modelled as a Boolean state variable[2]. The history of role-activations is relevant to the policy decision. The RM must maintain the value of all role-activations for the role admin of the previous state in order to enforce this rule. This is achieved by introducing a history variable $H_{\text{in,s,admin}}$ for all $s \in \mathcal{S}$, that stores a list containing the last two values of the subscripted variable. The RM now has to maintain the history by implementing the corresponding enforcement code as enf_{pre}.

$$enf_{pre} \mathrel{\widehat{=}} \forall s \in \mathcal{S} \bullet \\ H_{\text{in,s,admin}}[1], H_{\text{in,s,admin}}[0] \leftarrow H_{\text{in,s,admin}}[0], \text{in}(s, \text{admin})$$

This means that the RM will keep the history of all role activations for the role admin for the last state in the history variables $H_{\text{in,s,admin}}[1]$ and for the current state in $H_{\text{in,s,admin}}[0]$. Under the assumption that the variables remain stable unless assigned differently, it is possible to show that

$$\begin{aligned} enf_{pre} \mathrel{\widehat{=}} \text{for } s \text{ in } \mathcal{S} : \{ \\ & H_{\text{in,s,admin}}[1] := H_{\text{in,s,admin}}[0]; \\ & H_{\text{in,s,admin}}[0] := \text{in}(s, \text{admin}) \\ \} \end{aligned}$$

[2]More complex data-structures can affect the efficiency of the RM, however, the same concepts for the enforcement of rules apply.

is a refinement. The latter is closer to most implementation languages, as it does sequentialise the assignment of the history variables. Similar to the enforcement of state-dependent rules, the condition C_{autho} is then defined in terms of these history variables. In this case:

$$C_{\text{autho}} \mathrel{\widehat{=}} T \geq 1 \wedge \neg\, H_{\text{in,s,admin}}[1]$$

The concrete benefit of this approach is that only the *necessary* history is maintained — resulting is smaller histories.

Often rules are more complex, viz. they do not only reference the past value of a single variable, but may define the behaviour of variables over time. For example the rule

$$10 : (\text{done}(s, o, a_1)\,;\text{done}(s, o, a_2)) \mapsto \text{autho}(s, o, a_1)$$

expresses that if a subject s did 10 states in the past access a resource o in mode a_1 and subsequently in mode a_2 then it can access the same resource again in mode a_1. In this case the history of the control variables $\text{done}(s, o, a)$ must be kept for the last 10 states to determine outcome of the rule. This is implemented analogous to the previous case. Formally the rule states that if the past interval of length 10 can be decomposed into a prefix interval for which $\text{done}(s, o, a_1)$ holds and a suffix interval for which $\text{done}(s, o, a_2)$ holds then an authorisation for s to access o in mode a_1 can be derived. The condition C_{autho} can be expressed in terms of the history variables as:

$$T \geq 10 \wedge H_{\text{done},s,o,a_1}[10] \wedge \exists i \leq 10 \bullet H_{\text{done},s,o,a_2}[i]$$

The existential quantification is always bounded and can expressed as a disjunction. The transformation of a rule into enforcement code and corresponding access control conditions is straightforward given the formal semantics of the rules and can be automated. However, the resulting enforcement code is not necessarily the most efficient. This leads to the question of optimised enforcement.

C. Optimisation of Temporal Modalities

Often it is the case that only the existence of a past event or a test for an invariant condition is decisive for the outcome of a rule. So for example the rule

$$\Diamond\text{done}(s, \text{bPay}, \text{click}) \mapsto \text{autho}^{-}(s, \text{bPay}, \text{click})$$

prevents any subject from pressing the "pay" button twice. If the above transformation would be applied, we would obtain:

$$T : \Diamond\Diamond\text{done}(s, \text{bPay}, \text{click}) \mapsto \text{autho}^{-}(s, \text{bPay}, \text{click})$$

which is

$$T : \Diamond\text{done}(s, \text{bPay}, \text{click}) \mapsto \text{autho}^{-}(s, \text{bPay}, \text{click})$$

equivalent to:

$$T : (\text{true}\,;\text{done}(s, \text{bPay}, \text{click})) \mapsto \text{autho}^{-}(s, \text{bPay}, \text{click})$$

Resulting in a history variable for the control variable $\text{done}(s, \text{bPay}, \text{click})$ for all $s \in \mathcal{S}$ that grows linearly with

the enforcement time T and the condition for the negative authorisation rule to fire:

$$\exists i \leq T \bullet H_{\mathrm{done},s,\mathrm{bPay},\mathrm{click}}[i]$$

However, it is easy to show that once $\mathrm{done}(s, \mathrm{bPay}, \mathrm{click})$ has been true in a state σ'_i, $0 \leq i \leq T$, viz. the history variable $H_{\mathrm{done},s,\mathrm{bPay},\mathrm{click}}[T-i]$ is true, the condition holds for all subsequent states σ'_j, $i \leq j \leq T$. Instead of maintaining all the history it is therefore sufficient to have a single Boolean variable $H^{\dagger}_{\mathrm{done},s,\mathrm{bPay},\mathrm{click}}$ (per subject s) that indicates that the event has taken place. The variable is initialised to false. For this example the enforcement code would be:

$$enf_{pre} \mathrel{\widehat{=}} \forall s \in \mathcal{S} \bullet \mathrm{if}\ (\mathrm{done}(s, \mathrm{bPay}, \mathrm{click}))\ \mathrm{then}\ H^{\dagger}_{\mathrm{done},s,\mathrm{bPay},\mathrm{click}} \leftarrow \mathsf{true}$$

The access condition is only dependent on the value of this variable, resulting in a *constant* complexity of history maintenance and decision evaluation w.r.t. enforcement time. Similar optimisations can be made for dependencies on invariants where the rule has the form $\Box w \ \mapsto\ \mathrm{autho}(s, o, a)$. These optimisations can also be exploited in more complex settings as we show in the following using an access control rule that is common to version control systems such as CVS.

D. Example of a Version Control Rule

The rule in (10) results in a negative authorisation for the subject s to commit a file f on the CVS repository if another subject i did commit the file f to the repository and s did never subsequently update the file f. This guarantees that changes made by one author are not *accidentally* overwritten by another author.

$$\exists x \in \mathcal{S} \bullet x \neq s \wedge (\mathrm{done}(x, \mathrm{CVS}, \mathrm{commit}(\mathrm{f})) \wedge \Box \neg \mathrm{done}(s, \mathrm{CVS}, \mathrm{update}(\mathrm{f}))) \mapsto \mathrm{autho}^{-}(s, \mathrm{CVS}, \mathrm{commit}(\mathrm{f})) \quad (10)$$

Formally the rule defines the behaviour of the Boolean state variable $\mathrm{autho}^{-}(s, \mathrm{CVS}, \mathrm{commit}(\mathrm{f}))$ for all subjects $s \in \mathcal{S}$ and all files f in the repository over the states marked by m. Its value is true in state σ'_i if there is a suffix interval of $\sigma'_0 \ldots \sigma'_i$ such that the premise holds. For rule (10) this means there is a suffix interval $\sigma'_j \ldots \sigma'_i$ for $0 \leq j \leq i$ in which $\mathrm{done}(x, \mathrm{CVS}, \mathrm{commit}(\mathrm{f}))$, where subject x is different from s, is true in the initial state and $\mathrm{done}(s, \mathrm{CVS}, \mathrm{update}(\mathrm{f}))$ is false throughout the interval. We combine (10) with the decision rule in (11) to a simple policy.

$$\neg \mathrm{autho}(s, o, a)^{-} \ \mapsto\ \mathrm{autho}(s, o, a) \quad (11)$$

The decision rule states that an access request is granted, if no negative authorisation can be derived.

We show that it is not necessary to store the whole access history. The reason is that it is sufficient to find *one* interval in the left neighbourhood of the state for which the policy decision is made that satisfies the premise. For the rule in (10) the premise defines the *invariant* $\Box \neg \mathrm{done}(s, \mathrm{CVS}, \mathrm{update}(\mathrm{f}))$ and a condition of the initial state, viz. there is another subject x such that $\mathrm{done}(x, \mathrm{CVS}, \mathrm{commit}(\mathrm{f}))$. This is illustrated in Figure 8.

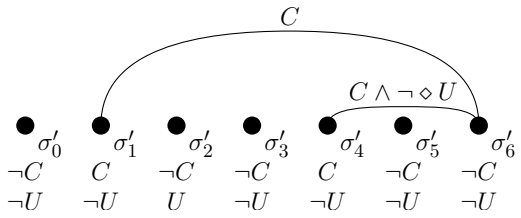

Fig. 8. Rule Optimisation

In Figure 8 we use the symbol C to denote the expression $\exists x \in \mathcal{S} \bullet \mathrm{done}(x, \mathrm{CVS}, \mathrm{commit}(\mathrm{f}))$ and U to denote $\mathrm{done}(s, \mathrm{CVS}, \mathrm{update}(\mathrm{f}))$. Let us assume that the policy decision is to be made for state σ'_6. Then only the suffix interval $\sigma'_4 \ldots \sigma'_6$ satisfies the premise of the rule. Another potential candidate would be the interval $\sigma'_1 \ldots \sigma'_6$, as C is true in state σ'_1, however the suffix does not satisfy the additional constraint $\Box \neg U$ (which is equivalent to $\neg \Diamond U$), because U is true in state σ'_2. It is easy to see that if there is a suffix interval

$$\sigma'_j \ldots \sigma'_{|\sigma'|} \models C \wedge \Diamond U$$

then for any suffix interval $\sigma'_i \ldots \sigma'_{|\sigma'|}$ where $i < j$:

$$\sigma'_i \ldots \sigma'_{|\sigma'|} \not\models C \wedge \neg \Diamond U$$

On the other hand, if there is an interval $\sigma'_j \ldots \sigma'_{|\sigma'|}$ such that

$$\sigma'_j \ldots \sigma'_{|\sigma'|} \models C \wedge \neg \Diamond U$$

then this is sufficient for the rule to fire and we are not interested whether there are other, longer suffix intervals that also satisfy the premise. Consequently, to determine whether the rule applies or not, we only need to consider the shortest suffix interval for which C holds in the initial state. It is then sufficient to determine whether $\neg \Diamond U$ holds for that interval or not.

The mechanism for the enforcement of the access control policy is implemented in enf_{pre} and is executed exactly once between two states in the interval σ'. To ensure that the policy is enforced correctly we have to define enf_{pre} and C_{autho} in such a way that E_{autho} (see (2)) holds over σ'. For the enforcement of rule (10) we introduce the boolean auxiliary variables $\mathrm{uptodate}(s, \mathrm{CVS}, \mathrm{f})$ that is true if subject s has an up to date revision of file f from the repository CVS. We initialise the auxiliary variables with the value true.

We know that the evaluation of the rule (10) is dependent on the event that a subject has committed a file f to the repository. We also determined that it is sufficient for the evaluation to check whether since the last commit event the invariant is satisfied or not. The latter is simple to check, as it is sufficient to find one state violating the invariant.

$$
\begin{aligned}
enf_{pre} \mathrel{\widehat{=}} & \forall s, \mathrm{f}, \mathrm{CVS} \bullet \\
& (\text{if } \mathrm{done}(s, \mathrm{CVS}, \mathrm{commit}(\mathrm{f})) \text{ then} \\
& \quad \forall i \in \mathcal{S} \setminus \{s\} \bullet \mathrm{uptodate}(i, \mathrm{CVS}, \mathrm{f}) \leftarrow \mathsf{false} \\
& \quad \wedge\ \mathrm{uptodate}(s, \mathrm{CVS}, \mathrm{f}) \leftarrow \mathsf{true} \\
&) \wedge \\
& \forall s, \mathrm{f}, \mathrm{CVS} \bullet \\
& (\text{if } \mathrm{done}(s, \mathrm{CVS}, \mathrm{update}(\mathrm{f})) \text{ then} \\
& \quad \mathrm{uptodate}(s, \mathrm{CVS}, \mathrm{f}) \leftarrow \mathsf{true} \\
&)
\end{aligned}
$$

If a subject commits a new revision of a file f all other subjects do not have an up to date revision, and the version of s can be considered up to date. If a subject did perform an update on file f then this subject has an up to date revision of f. The auxiliary variable $\mathrm{uptodate}(s, \mathrm{CVS}, \mathrm{f})$ is set to false if at some point in the past the a subject different to s did commit the file f. It can only be set to true if s updates the file f. The condition for the rule to fire is therefore solely dependent on the auxiliary variable $\mathrm{uptodate}(s, \mathrm{CVS}, \mathrm{f})$.

Having shown how the formal semantics of rules can be used to derive the enforcement code we now address the enforcement of policy compositions.

E. Enforcing Policy Compositions

Policies in the model can be composed along a temporal and a structural axis. We concentrate here on the discussion of temporal compositions. A temporal composition of two policies defines the dynamic transition from one policy to another, triggered by the observation of an event. Consider the policy:

$$\langle \mathrm{event} \rangle P \,;\, Q$$

as a simple representative of temporal composition. *Unless* the event occurs policy P is enforced, then the policy Q. As we established previously the event handling and management of histories is implemented in enf_{pre}. We take a similar approach to handle the change of policies.

For the above example composition the RM must be able to distinguish between the enforcement of P and Q, viz. whether the event occurred or not. For this purpose we introduce an auxiliary variable $H^{\dagger}_{\mathrm{event}}$ similar to the history variables. For the composition we define enf_{pre} then as:

$$
\begin{aligned}
enf_{pre} \mathrel{\widehat{=}} & \text{if (event) then } H^{\dagger}_{\mathrm{event}}, T \leftarrow \mathsf{true}, 0; \\
& \text{if } (H^{\dagger}_{\mathrm{event}}) \text{ then} \\
& \quad \text{derived code for policy } P \\
& \text{else} \\
& \quad \text{derived code for policy } Q
\end{aligned}
$$

This shows that deriving enforcement code is compositional and emphasises another advantage of the policy specification approach: The explicit identification of policy changes leads to a reduced complexity in the enforcement. The reason for this is that only the currently relevant part of the overall policy is evaluated (e.g. P) whereas rules contained in the other parts (e.g. Q) do not affect the enforcement. Note also that with the change of policy the enforcement time T is reset, ensuring that the derived enforcement code corresponds with the semantics of the policy composition.

VIII. Related Work

There is an agreement in the policy community that todays complex protection requirements require policies to be stateful, viz. policy decisions depend on the current state of the system. For some systems this state is an explicit part of the trusted computing base, e.g. stack-based models [11], Role-Based Access Control [12] or Multi-Level Security [13]. For others state can be defined in form of mutable attributes [14] as part of access control policies [15]. Other models allow to base policy decisions on the execution history [6], where policy decision can depend on the system behaviour that was observed in the past. This has been shown to be more expressive than some of the more traditional mechanisms such as stack inspections in [6], and is now supported by many policy languages, see e.g. [16] for an overview. A standard example of a stateful policy is the Chinese Wall Policy [17].

The enforcement of history based policies can prove to be a large overhead in the administration of past events and evaluation of policies that depend on them. Gamma et.al. [18] propose a form of meta-policies that allow to purge the history to improve scalability. However, it is not clear how the semantics of the policy is maintained, given that information that is relevant to the policy decision can be lost. The approach of minimising the history information based on the policies has been adopted in [19] where the policy is compiled into a concrete enforcement code. While we agree that automation is key to the success of policy-managed systems, it is not clear how one certifies that the result of the compilation corresponds to the semantics of the policy.

The potential of temporal logic specifications has been noted recently by Calo et.al. [20] and has been successfully applied in areas where dynamic properties are relevant e.g. [21]. Recently the UCON model has been formalised by Zhang [22] using TLA [23]. Most notably is the approach of Chomicki [24] that uses Past Time First Order Temporal Logic to express and optimise integrity checks in database systems. Both Ribeiro and Chomicki noted that the optimisation of the stored history causes problems when new policies are enforced, as their relevant history may not have been recorded. We do not solve this problem either, however, our approach is truthful to the semantics of the policy language, viz. even under policy changes the enforcement is correct w.r.t. the enforcement properties. This is mainly due to the scoping that is achieved by the ITL Chop operator for sequential composition.

The enforcement of policy languages has been studied by Schneider [25] where security policies are enforced program monitors using security automata. Ligatti et.al. [26] extended this work and showed how a certain kind of obligations can

be enforced. However, this work assumes that the policy is already available as an automata, thus ignoring important aspects such as the maintenance of event histories.

IX. Conclusions

In this paper we have presented a formal model of a reference monitor for the enforcement of dynamically changing policies [8], [9]. To close the gap between the more abstract policy specification and the concrete implementation of enforcement code we shown how both specifications can be linked using temporal projection. We formally defined enforcement-properties that express the intuitive notion of *correct* enforcement at a high abstraction level and showed how policy rules can be transformed into enforcement code that guarantees the compliance of the system with these properties.

The focus of this paper was the enforcement of access control rules contained in the policy. This is divided into two parts: *i)* the maintenance of the required history for temporal dependent rules and *ii)* the definition of a corresponding access control condition using this history information. We also showed how the formal model can be used to optimise the history and the evaluation of the access control condition. The strength of our approach is that it allows to formally prove that the derived enforcement code satisfies the enforcement properties w.r.t. a specific policy. Additionally the derived code is at a sufficiently low abstraction level to be readily implementable in conventional programming languages. We envision this technique to be used in resource-limited settings, where guarantees on the required storage capacity are needed or where the timeliness of policy decision making is critical.

Related to the presented work we have also shown how integrity policies can be enforced using similar mechanisms to the described in combination with a simple commit protocol. In this case the success or failure of the request is dependent on an additional integrity check (6. in Figure 5). We plan to extend this work to provide a policy framework that allows for the automated derivation of enforcement mechanisms. The result of this being a *certifiable* reference monitor that can be integrated in systems with relative ease.

Acknowledgement

This work has been undertaken as part of DIF-DTC project 12.5.1 (see `http://www.difdtc.com`).

References

[1] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, pp. 333–360, 1994. [Online]. Available: citeseer.ist.psu.edu/sloman94policy.html

[2] ISO/IEC, "ISO/IEC 10181-3:1996 Information technology – Open Systems Interconnection – Security frameworks for open systems: Access control framework," March 2006. [Online]. Available: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=18199

[3] M. Abadi, "Logic in Access Control," in *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, vol. 15. Ottawa, Canada: IEEE Computer Society Press, June 2003, pp. 228–233. [Online]. Available: citeseer.ist.psu.edu/article/abadi03logic.html

[4] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and Semantics of a Decentralized Authorisation Language," Microsoft Research, Roger Needham Building 7 J.J. Thompson Avenue, Cambridge, CB3 0FB, UK, Tech. Rep., September 2006.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*. New Jersey: Prentice Hall, 2002.

[6] M. Abadi and C. Fournet, "Access control based on execution history," in *10th Annual Network and Distributed System Symposium (NDSS'03)*. Reston, Virginia, USA: The Internet Society, February 2003, pp. 1–15. [Online]. Available: citeseer.ist.psu.edu/abadi03access.html

[7] H. T. Janicke, "The Development of Secure Multi-Agent Systems," Ph.D. dissertation, De Montfort University, February 2007.

[8] H. Janicke, A. Cau, F. Siewe, H. Zedan, and K. Jones, "A Compositional Event & Time-based Policy Model," in *Proceedings of POLICY2006, London, Ontario, Canada*. London, Ontario Canada: IEEE Computer Society, June 2006, pp. 173–182.

[9] F. Siewe, "A Compositional Framework for the Development of Secure Access Control Systems," Ph.D. dissertation, Software Technology Research Laboratory, Department of Computer Science and Engineering, De Montfort University, Leicester, 2005.

[10] B. Moszkowski, "Compositional Reasoning about Projected and Infinite Time," in *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*. Fort Lauderdale, Florida: IEEE Computer Society Press, Novenmber 1995, pp. 238–245.

[11] L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security: Architecture, API Design and Implementation*, 2nd ed. Addison-Wesley Professional, 2003, iSBN: 0201787911.

[12] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[13] L. J. LaPadula and D. E. Bell, "Secure Computer Systems: Mathematical Foundations," MITRE Technical Report 2547, Tech. Rep., 1973.

[14] J. Park, X. Zhang, and R. S. Sandhu, "Attribute Mutability in Usage Control." in *Proceedings of IFIP TC11/WG 11.3 Eighteenth Annual Conference on Data and Applications Security*, C. Farkas and P. Samarati, Eds. Sitges, Catalonia, Spain: Kluwer, July 2004, pp. 15–29.

[15] R. Sandhu and J. Park, "The UCON$_A BC$ usage control model," in *Proceeding of the Second International Workshop on Mathematical Method, Models and Architectures for Computer Networks Security*, 2003.

[16] A. K. Bandara, E. C. Lupu, and M. Sloman, "Policy-Based Management," in *Handbook of Network and System Administration*, M. Burgess and J. Bergstra, Eds. Elsevier, 2007, ch. Policy Based Management.

[17] D. Brewer and M. Nash, "The Chinese Wall Policy," in *IEEE Symposium on Research in Security and Privacy*. Oakland, California, USA: IEEE, May 1989, pp. 206–214.

[18] P. Gama, C. N. da Cruz Ribeiro, and P. Ferreira, "A Scalable History-based Policy Engine," in *Seventh IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2006)*. IEEE Computer Society, June 2006, pp. 100–109.

[19] C. N. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, "SPL: An access control language for security policies with complex constraints," in *Network and Distributed System Security Symposium (NDSS'01)*. San Diego, California: The Internet Society, February 2001, pp. 1–19.

[20] S. Calo and J. Lobo, "A Basis for Comparing Characteristics of Policy Systems," in *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY2006)*. London, Ontario, Canada: IEEE Computer Society, June 2006, pp. 183–192.

[21] T. Mossakowski, M. Drouineaud, and K. Sohr, "A temporal-logic extension of role-based access control covering dynamic separation of duties," in *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003)*. Cairns, Queensland, Australia: IEEE Computer Society, July 2003, pp. 83–90.

[22] X. Zhang, F. Parisi-Presicce, R. S. Sandhu, and J. Park, "Formal model and policy specification of usage control." *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 4, pp. 351–387, 2005.

[23] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[24] J. Chomicki, "Efficient checking of temporal integrity constraints using bounded history encoding," *ACM Trans. Database Syst.*, vol. 20, no. 2, pp. 149–186, 1995.

[25] F. B. Schneider, "Enforceable Security Policies," *ACM Transactions on*

Information and System Security, vol. 3, no. 1, pp. 30–50, February 2000.

[26] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies." *Int. J. Inf. Sec.*, vol. 4, no. 1-2, pp. 2–16, 2005.