Timed Constraint Programming: A Declarative Approach to Usage Control

Radha Jagadeesan*
Will Marrero†
Corin Pitcher‡
DePaul University
Chicago, IL 60604

Vijay Saraswat§
IBM Research
Yorktown Heights, NY

ABSTRACT

This paper focuses on policy languages for (role-based) access control [14, 32], especially in their modern incarnations in the form of trust-management systems [9] and usage control [30, 31]. Any (declarative) approach to access control and trust management has to address the following issues:

- Explicit denial, inheritance, and overriding, and
- History-sensitive access control

Our main contribution is a policy algebra, in the timed concurrent constraint programming paradigm, that uses a form of default constraint programming to address the first issue, and reactive computing to address the second issue.

The policy algebra is declarative — programs can be viewed as imposing temporal constraints on the evolution of the system — and supports equational reasoning. The validity of equations is established by coinductive proofs based on an operational semantics.

The design of the policy algebra supports reasoning about policies by a systematic combination of constraint reasoning and model checking techniques based on linear time temporal-logic. Our framework permits us to perform security analysis with dynamic state-dependent restrictions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*

General Terms

Languages, Security, Verification

*rjagadeesan@cs.depaul.edu. Supported in part by NSF 0430175.
†wmarrero@cs.depaul.edu
‡cpitcher@cs.depaul.edu
§vsaraswa@us.ibm.com

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

Keywords

Constraints, Reactive Systems, Usage Control, Role-Based Access Control, Trust Management

1. INTRODUCTION

There is a recent resurgence of interest in research into policy languages. This interest is motivated in part by the loopholes and the drawbacks in recently proposed standards for policy languages in a variety of domains, (e.g., see [17] for an analysis of a language for software licenses, and [2, 6] for an analysis of a policy language for privacy in enterprises).

In the rest of this introduction, we consider examples to motivate the two main issues underlying usage control; examine related work and place our contributions in context.

Explicit denial, inheritance and overriding. We first present a simplified account of the Windows access control list (ACL) mechanism [39]. Recall that an ACL for an object such as a file is composed of access control entries (ACEs) representing access rights granted or denied to users.

Example 1.1 *Figure 1 is an excerpt of a database of ACL rules using the syntax of [24]. The order of the definitions is irrelevant. We take the set of principals to consist of the filesystem `FS`, the local security authority `LSA`, and a set of users. The LSA's local names are groups that denote sets of users. The filesystem's local names also include the positive or negative access rights parameterized by the absolute path of a file or directory. We consider the `READ_CONTROL` permission, which authorizes reading a file or directory. The filesystem's local name `FS.READ_CONTROL+(path)` denotes the set of users that appear in positive ACEs. The negative ACEs are denoted by `FS.READ_CONTROL-(path)`. The names `FS.READ_CONTROLI+(path)` and `FS.READ_CONTROLI-(path)` are similar but refer to ACEs inherited from the parent of `path`.*

When read access to a file object `path` is requested by a user s, the security monitor carries out the procedure described in Figure 2.

Following existing research, including the delegation logic [22] and the RT framework [24], one can read the "positive" rules from the ACL model as axioms for permitted delegation. For example, the fourth rule for read-control in Figure 1 is read as "an access-check for read of the patients sub-directory is satisfied by membership in the role `LSA.Doctors`". The "negative" rules are used for *explicit denial* to achieve concise descriptions: eg., the first rule with `FS.READ_CONTROL-` on

Figure 1 Excerpt from Windows ACL model

```
LSA.Administrators <- Administrator
LSA.Doctors <- Alice
LSA.Doctors <- Bob
LSA.Suspended <- Bob

FS.READ_CONTROL+("\") <- LSA.Administrators
FS.READ_CONTROL+("\Docs and Settings\Alice") <- Alice
FS.READ_CONTROL-("\Patients") <- LSA.Suspended
FS.READ_CONTROL+("\Patients") <- LSA.Doctors
FS.READ_CONTROLI+("\Patients") <- FS.READ_CONTROL+("\")
FS.READ_CONTROLI+("\Patients") <- FS.READ_CONTROLI+("\")
FS.READ_CONTROLI-("\Patients") <- FS.READ_CONTROL-("\")
FS.READ_CONTROLI-("\Patients") <- FS.READ_CONTROLI-("\")
```

Figure 2 The order of evaluation in evaluating access requests from user s to file object `path`

1. If s is in `FS.READ_CONTROL-(path)`, then deny the request, otherwise continue.
2. If s is in `FS.READ_CONTROL+(path)`, then allow the request, otherwise continue.
3. If s is in `FS.READ_CONTROLI-(path)` (the negative ACEs inherited from the parent), then deny the request, otherwise continue.
4. If s is in `FS.READ_CONTROLI+(path)` (the positive ACEs inherited from the parent), then allow the request, otherwise continue.
5. Deny the request.

the left denies access to those suspended from duty. In addition, as reflected in the last four rules of Figure 1 and in the algorithm in Figure 2, the Windows ACL uses *inheritance* and *overriding* to minimize repetition of access control lists (ACLs) on hierarchically-structured collections. These rules allow a child object to inherit ACLs from its parent and to override them locally.

History and temporal aspects. Consider the following example drawn verbatim from [44].

Example 1.2 ([44]) *Suppose in a DRM application with limited number of simultaneous usages, an object o can only be accessed and used by 10 users at a time. Each new access request is allowed. We assume that there is only one access generated from a single user. If the number of users accessing the object is 10, then one existing user's ongoing access is revoked when a new request is generated. There can be different policies to determine which users ongoing access must be stopped as follows. (a) Revocation by start time: the longest usage will be revoked. (b) Revocation by idle time: the usage with the longest idle time is revoked (c) Revocation by total usage time: the user with the longest accumulating usage time will be revoked.*

What is key in this example, for our purposes, is that the past interactions with the DRM application determine future responses of the system.

Such temporal requirements are further illustrated by Brewer and Nash's Chinese Wall security policy [12]. This policy is a mandatory access control model that prevents an employee passing data, maliciously or accidentally, from one client to another client when there would be a conflict of interest between the two clients, e.g., when the clients are engaged in legal action but are both represented by lawyers from the same firm. The Chinese Wall model is history-sensitive in the sense that access control decisions are based on the ownership of the data previously read by the user.

Current approaches. The traditional approach to formalizing such access-control languages is logic-based. These are usually in some fragment of many-sorted first-order predicate logic, e.g., [16], with sorts for roles and time, and with suitable restrictions to ensure effective computability. Often, these restrictions are set up to ensure that there is a compilation down to an efficient execution engine, e.g., based on variants and extensions of Datalog. Such research includes the delegation logic [22], the RT framework [24] approach to trust-management, and compositional approaches to access-control [11, 42, 43] that compile down to logic programs. Implementation techniques help in achieving efficient execution [4]. Constraint (logic) programming appears as a tool in some research with the role of constraint systems being to axiomatize the domain-specific reasoning of interest — such research includes the use of Datalog with constraints [23] and the use of constraint programming to model access control [5] and attribute based access control [41].

These approaches suffer some shortcomings with respect to the two key issues.

First, if the underlying formalism is Datalog without explicit negation, one is usually operating under the assumption that anything not explicitly permitted is prohibited. [16] argues that it is unclear that this is the case for policy makers. Stratifying the execution of Datalog programs leads to the ability to use negation in restricted circumstances. A full treatment of negations requires that we work with Datalog with explicit negation [21] and work with a closed world view, i.e., all facts not provable are considered to be not available. The Datalog view, even with negation, does not permit convenient encoding of *inheritance* and *overriding*, features that are crucial in some applications. For example, we have seen that Windows uses these features to efficiently realize access control lists (ACLs) on hierarchically-structured collections.

Secondly, (logical) time is delegated to a sort and treated implicitly. We feel that this is inappropriate in modern usage control systems which are fundamentally dependent on the past evolution of the system. The state sensitive aspect of access control is now becoming well-accepted — e.g., see [1], temporal extensions to role-based access control (RBAC) [8] and state-transition approaches to trust management [13].

Our approach. Our solution to the first problem is based on moving to (concurrent) constraint programming [20, 36] as the underlying declarative basis. In such a setting, there is a natural 3-valued interpretation since the results of a query to a constraint system can be yes, no and don't know. Our policy algebra provides a way to write permitting or denying policies as well as a mechanism to override policies based on a restricted notion of defaults. Our policy combination operators are logical operators in such a 3-valued world, and permit the prescription of methods to detect and resolve conflicts via operations on 3-valued truth values.

Our solution to the second problem is based on the approach to reactive systems inspired by timed concurrent constraint programming [34]. Recall that reactive systems are those that react continuously with their environment at a rate controlled by the environment. Execution in a reactive system proceeds in bursts of activity. Thus, a reactive system has a logical notion of time: at each time instant, the environment stimulates the system with an input, obtains a response, and may then be inactive (with respect to the system) for an arbitrary period of time before initiating the next burst. Existing research on temporal specification languages for usage control policies [44] and access control policies [37] demonstrates the expressiveness gained by explicitly incorporating such a logical notion of time.

What is new in our approach is an integrated executable and declarative framework that incorporates both time and the domain-specific reasoning (via constraint systems) required for dealing with roles and other important abstractions. Thus, building on the line of algebraic approaches to policy composition [11, 42, 43], and in contrast to other executable state machine approaches [28, 40, 38, 13], we describe a declarative algebra of temporal operators.

Implementation and Analysis. The policy algebra is a domain-specific language in the timed constraint programming paradigm. This opens up the possibility of adapting general-purpose programming language techniques to achieve domain-specific objectives.

`jcc` [35] is a Java library, available from SourceForge, that realizes timed default constraint programming [34]. There is a translation of our policy algebra into `jcc`. We are in the process of implementing this translation. This implementation will enable the use of general programming language techniques to inline the reference monitors into applications. In this extended abstract, we elide details of the translation and implementation.

When restricted to finitely many principals and resources, the explicit relationship of our framework to reactive systems enables us to build upon and reuse extant specification and verification machinery. We illustrate the utility of such an analysis by factoring the safety and availability analysis of [25] into two orthogonal and well-studied pieces of research: (a) the analysis of Datalog programs developed in [25] and (b) LTL model-checking to handle assumptions about the interaction with the environment. More importantly, our framework permits us to perform security analysis with dynamic state-dependent restrictions — this was left as an open problem in [25].

The rest of the paper. We begin with an informal introduction to the language in section 2. In this section, we illustrate the language with a variety of examples. In section 3, we provide an operational semantics and define bisimilarity, which is a congruence. We use bisimulation arguments to provide the basic rudiments of equational reasoning. We establish the basic framework needed for the analysis of finite-state policies and provide examples of analysis with dynamic state-dependent restrictions that can be carried out in our framework. Finally, we conclude by placing our work in the global context of declarative techniques.

2. AN OUTLINE OF OUR APPROACH

The (concurrent) constraint (`cc`) programming paradigm replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. Computation progresses by accumulating constraints in the store, and by checking whether the store entails constraints. In this section, we describe a domain-specific policy algebra in the `cc` paradigm. Our approach to a timed language of policies is inspired by declarative approaches to reactive computing [18, 7, 15] and is a domain-specific sub-language of Timed Default Concurrent Constraint programming [34].

2.1 Syntax

The syntax of the policy algebra is described in Figure 3.

Figure 3 Syntax of temporal policy algebra

Value grammar:

K ::=	`true`	Positive response (grant)
|	`false`	Negative response (deny)
|	$\bot$	No response

Body grammar:

$P, Q ::= K$	
| $\mathtt{not}(P)$	
| P `relop` Q	`relop` $\in$ {`and`, `or`, `def`, `left`} Boolean policy combination
| `new` x `in` P	Local variable creation
| $f(\vec{t})$	Invoke policy with terms $\vec{t}$
| `if` a `then` P `else` Q	Ask tokens
| `if` P `then` Q_1 `else` Q_2	Query Policy
| `hence` P	
| `time` P `on-present` a	
| `time` P `on-absent` a	
| $\mathtt{next}(\mathtt{tell}(a))$	

Policy declarations have the form:

$$f(\vec{x}) :: P$$

In such a policy declaration, the free variables of P must be contained in $\vec{x}$.

We use a for constraint tokens and $\vec{t}$ for terms.

2.2 Execution model

The programs in our policy algebra are reactive, meaning that a program may interact with its environment in a sequence of discrete time steps. At each time step, the environment provides an input that can be either:

- a query to a policy, or
- other input events which may not need a grant or deny response but that may affect policy queries in the future

An example of the first kind of environment stimulus is a subject requesting access to an object via a policy invocation. An example of the second kind of environment stimulus are $\mathtt{endaccess}(s, o, r)$ events that represent a notification from subject s of the end of its access to object o that required right r. The latter kind of events arise inevitably in history-sensitive access control.

The arrival of a new input event and its processing advances the conceptual logical discrete clock. In response to the input, execution is carried out to quiescence and results in two kinds of information:

- A response to the stimulus. The response can be any of three values — grant (`true`), deny (`false`), or undefined ($\perp$).
- A "continuation" to be executed at future time instants. The continuation policy enforces future obligations and carries the state required to enforce the policy — it does not directly respond to policy queries *per se*.

2.3 Constraint systems

Communication in concurrent constraint programming is based on a generic, parametric notion of first-order pieces of partial information: first-order because the constraints involve *variables* over some underlying domain of discourse, partial because constraints do not necessarily completely determine the values that variables take. All `cc` languages are built generically over constraint systems [33, 36].

A constraint system is a multi-sorted (intuitionist) first order theory with equality that axiomatizes domain-specific reasoning. A constraint system is a system of partial information, consisting of a set of primitive constraints (first-order formulae) or *tokens* D, closed under conjunction and existential quantification, and an inference relation (logical entailment), denoted by $\vdash$, that relates tokens to tokens. We use $a, b, \ldots$ to range over tokens. For example, constraints can be expressions of the form $\mathtt{X} \geq \mathtt{Y}$, or "the sum of the weights of the vehicles on the bridge must not exceed a given limit". Constraints come equipped with their own entailment relation, that determines which pieces of information (e.g., $\mathtt{X} \geq \mathtt{Z}$) follow from other pieces of information (e.g. $\mathtt{X} \geq \mathtt{Y}$ and $\mathtt{Y} \geq \mathtt{Z}$). The formal definition of constraint systems is by now standard — we refer the reader to [36].

Traditional examples of such systems include the system Herbrand (underlying logic programming), Finite Domains (FD) [19], and `Gentzen` [34]. The constraint systems relevant to the domain of interest of this paper include Tree-Domains [23] and the Datalog constraint system [23].

Example 2.1 The Herbrand constraint system. *Let L be a first-order language L with equality. The tokens of the constraint system are the atomic propositions. Entailment is specified by Clark's Equality Theory, which include the usual entailment relations that one expects from equality. Thus, for example, $f(X,Y) = f(A, g(B,C))$ must entail $X = A$ and $Y = g(B,C)$.*

Example 2.2 The Tree Domain *For specification of hierarchies such as role hierarchies and DNS names, the tree domain is useful. Each constant of a tree domain takes the form $\langle a_1, \ldots a_n\rangle$ that represents a path in a labelled tree with each a_i being the string on an edge in the path. A primitive constraint is of the form $x\ \mathtt{relop}\langle a_1, \ldots a_n\rangle$ where $\mathtt{relop} \in \{<, \leq, \prec, \preceq\}$. $x < \mathtt{relop}\langle a_1, \ldots a_n\rangle$ means that x is a child of the node $\langle a_1, \ldots a_n\rangle$, and $x \prec \langle a_1, \ldots a_n\rangle$ means that x is a descendant of the node $\langle a_1, \ldots a_n\rangle$. For example, $x \prec \langle \mathtt{edu}, stanford\rangle$ captures addresses in the domain* `stanford.edu`.

Example 2.3 Datalog Constraints *Datalog clauses are already in the form permissible as rules in a constraint system. A constraint Datalog program with constraints in several domains can be evaluated in polynomial time when the constraint domains in question are also polynomially decidable [23]. When convenient we use RT notation from [23] such as:*

```
FS.READ_CONTROL+("\Patients") <- LSA.Doctors
```

to represent constraints and $\mathtt{isMember}(B, A, R)$ to test the membership of a principal B in the local name R of a second principal A. In this paper the constraints corresponding to RT rules are used only as tell-constraints, i.e., we only use them for deductions and do not query if they are in the store. Formally, they do not appear on the right-hand side of the entailment relation. Constraints such as $\mathtt{isMember}(\mathtt{user}, \mathtt{FS}, \mathtt{READ\_CONTROL}-(\mathtt{path}))$ are used to query the store. This query can yield a value `true` if the rules suffice to establish this membership or a value $\perp$ if the membership does not follow from the rules.

In our programming examples, we will usually work with a constraint system that includes **Datalog Constraints**.

2.4 The Untimed language

Before presenting the examples without temporal features, we give an informal overview of the language design to place the policy algebra in the context of (concurrent) constraint programming languages.

The basic context of the policy algebra is 3-valued logic. A query to a policy can yield three values: grant, deny, and a third value $\perp$ coding "undefined" and arising from the absence of a response from a policy.

In our policy framework, we have several variants of the basic parallel composition operation of concurrent constraint programming. All these variants impose the constraints of both P and Q. The differences between the variants is in the boolean operation that they perform to combine the responses (i.e., grant, deny, undefined) from P and Q. These boolean operators have the expected meaning as characterized in the following truth tables, due to Lukasiewicz.

$$
\begin{aligned}
b_1\ \mathtt{or}\ b_2 &= \begin{cases} \mathtt{true}, \text{ at least one of } b_i \text{ is } \mathtt{true} \\ \mathtt{false}, \text{ both } b_i \text{ are } \mathtt{false} \\ \perp, \text{ otherwise} \end{cases} \\
\mathtt{not}\ b &= \begin{cases} \mathtt{false}, \quad b \text{ is } \mathtt{true} \\ \mathtt{true}, \quad b \text{ is } \mathtt{false} \\ \perp, \quad b \text{ is } \perp \end{cases} \\
b_1\ \mathtt{and}\ b_2 &= \begin{cases} \mathtt{false}, \text{ at least one of } b_i \text{ is } \mathtt{false} \\ \mathtt{true}, \text{ both } b_i \text{ are } \mathtt{true} \\ \perp, \text{ otherwise} \end{cases} \\
b_1\ \mathtt{left}\ b_2 &= b_1 \\
b_1\ \mathtt{def}\ b_2 &= \begin{cases} b_1, & b_1 \neq \perp \\ b_2, & b_1 = \perp \end{cases}
\end{aligned}
$$

For each boolean operation, we have a variant of the parallel composition operator.

- The version P `left` Q discards the response from Q and returns the value from P. A typical use of this combinator in our algebra is to use Q as a continuation for the future with responses to policy queries, if any, coming from P.

- We have binary operators on policies corresponding to the boolean operations — e.g., a binary combinator P or Q imposes the constraints of both P and Q but also yields a response to an access query that is the disjunction of the responses of the individual policies P and Q.
- An interesting combinator in this flavor is an overriding combinator P def Q that overrides the results given by Q by those provided by P. In particular, the results of Q are used if P *does not* provide a response to the access request.

The boolean policy combinators satisfy several expected distributivity and DeMorgan properties induced by the laws satisfied by the corresponding boolean operations— see Figure 4 for a sample. The equality, $\equiv$, in these equations is bisimilarity that is defined and shown to be a congruence in section 3.

Figure 4 Equational laws for Boolean combinators

$$\begin{array}{rcl}
P_1 \texttt{ and } (P_2 \texttt{ or } P_3) & \equiv & (P_1 \texttt{ and } P_2) \texttt{ or } (P_1 \texttt{ and } P_3) \\
P_1 \texttt{ or } (P_2 \texttt{ and } P_3) & \equiv & (P_1 \texttt{ or } P_2) \texttt{ and } (P_1 \texttt{ or } P_3) \\
(P) \texttt{ def } (P_1 \texttt{ and } P_2) & \equiv & (P \texttt{ def } P_1) \texttt{ and } (P \texttt{ def } P_2) \\
(P) \texttt{ def } (P_1 \texttt{ or } P_2) & \equiv & (P \texttt{ def } P_1) \texttt{ or } (P \texttt{ def } P_2) \\
(P_1 \texttt{ and } P_2) \texttt{ def } (P) & \equiv & (P_1 \texttt{ def } P) \texttt{ and } (P_2 \texttt{ def } P) \\
(P_1 \texttt{ or } P_2) \texttt{ def } (P) & \equiv & (P_1 \texttt{ def } P) \texttt{ or } (P_2 \texttt{ def } P)
\end{array}$$

The ask query (if a then P else Q) imposes the constraints of P provided the store entails the constraint a; it imposes the constraints of Q if the store does not entail the constraint a. In temporal terms, this query operation is instantaneous. In policy terms, this represents the restriction of the policy P to the individuals who satisfy a. We use if a then P as shorthand for if a then P else $\perp$. Thus, if a then P does not yield a response — the response is $\perp$ or undefined — if the store does not entail a. The ask combinator satisfies several expected distributivity laws — see Figure 5.

Figure 5 Equational laws for the ask combinator

$$\begin{array}{rcl}
\texttt{if } a \texttt{ then } (P_1 \texttt{ or } P_2) & \equiv & (\texttt{if } a \texttt{ then } P_1) \texttt{ or } (\texttt{if } a \texttt{ then } P_2) \\
\texttt{if } a \texttt{ then } (P_1 \texttt{ and } P_2) & \equiv & (\texttt{if } a \texttt{ then } P_1) \texttt{ and } (\texttt{if } a \texttt{ then } P_2) \\
\texttt{if } a \texttt{ then not} P & \equiv & \texttt{not}(\texttt{if } a \texttt{ then } P) \\
\texttt{if } a \texttt{ then } (P_1 \texttt{ def } P_2) & \equiv & (\texttt{if } a \texttt{ then } P_1) \texttt{ def } (\texttt{if } a \texttt{ then } P_2)
\end{array}$$

Finally, the untimed combinators include new variables that behave as expected, as illustrated by the laws of Figure 6.

Coding the Windows ACL. To model the Windows ACL mechanism, we further assume that we are working with

Figure 6 Equational laws for the new combinator

$$\begin{array}{rcl}
\texttt{new } x \texttt{ in new } y \texttt{ in } P & \equiv & \texttt{new } y \texttt{ in new } x \texttt{ in } P \\
\texttt{new } x \texttt{ in } P & \equiv & P,\ x \text{ not free in } \texttt{P}
\end{array}$$

Figure 7 The Windows Read File Access Policy program

```
CheckPos(user,path,XR) ::
      if isMember(user,FS,XR(path)) then
         grant

CheckNeg(user,path,XR) ::
      not(CheckPos(user,path,XR))

ReadFileAccess(user,path) ::
     CheckNeg(user,path,READ_CONTROL-) def
      (CheckPos(user,path,READ_CONTROL+) def
        (CheckNeg(user,path,READ_CONTROLI-) def
          (CheckPos(user,path,READ_CONTROLI+) def
             deny)))
```

the constraint system **Datalog Constraints**. The policies CheckPos, CheckNeg, and ReadFileAccess of Figure 7 realize the access control policy of the Windows filesystem. We discuss this program bottom-up to introduce the programming elements of the policy algebra.

CheckPos only performs a membership check. For example, in the case of a read request, membership is tested in various READ_CONTROL ACLs. Inheritance of ACLs (based on the hierarchy of the filesystem, in practice) can be encoded using RT statements as shown in Figure 1 and so occurs in the constraint system. As discussed earlier, the membership check yields a true value if membership is deducible from the Datalog program representing the RT rules, and undefined or unknown if the membership is not deducible. We use grant and deny as syntactic sugar for true and false respectively in examples.

Overridable defaults. The overall policy ReadFileAccess uses a default mechanism to specify overridable policies. In P_1 def P_2, the response of P_2 is used only if P_1 does not yield a defined response to an access request[1].

The primary use of this combinator in our context is to establish overridable defaults. The default combinator establishes the relative priority of different policy pieces by their static positions in the program text w.r.t. the def combinator. For example, in the ReadFileAccess policy

[1] A more general variant of the default combinator is an n-ary max combinator [28] with the following intended truth table semantics. For a vector $\vec{b} = \langle b_1, \ldots, b_n \rangle$ let $\#_{\texttt{true}}(\vec{b})$ (resp. $\#_{\texttt{false}}(\vec{b})$ be the number of true(resp. false) entries.

$$\max(\vec{b}) \quad = \quad \begin{cases} \texttt{true}, \ \#_{\texttt{true}}(\vec{b}) > \#_{\texttt{false}}(\vec{b}) \\ \texttt{false}, \ \#_{\texttt{false}}(\vec{b}) > \#_{\texttt{true}}(\vec{b}) \\ \perp, \ \text{otherwise} \end{cases}$$

In this extended abstract, we do not explore this further because of space constraints.

the outermost policy `CheckNeg(user,path,READ-CONTROL-)` has highest priority, overriding inherited ACLs checked in the policy `CheckPos(user,path,READ-CONTROLI+)`, which in turn has priority higher only than the policy that always denies. This is an example of a “closed policy”, where the “closed” assumption is encoded formally in our policy program by the innermost `deny`. Replacing this by a `grant` would have yielded an open policy encoding the specification that replaces line (5) of Figure 2 by “grant the request”.

An example from [16] illustrates the possibilities that arise in the interaction between the three-valued nature of constraint entailment and the default combinator.

Example 2.4 *The policy says that anyone without bad credit is granted access to a resource. In this example, we are using a unary policy* `badCredit` *to capture whether an individual has bad credit. We write two variants below.*

The policy `P1` *given by:*

```
P1(s) :: if not(badCredit(s)) then grant
```

is undefined for individuals whose credit cannot be established. So, `P1(s) def deny` *grants access to anyone who can be proved to not have bad credit, and denies access to everyone else.*

In contrast, the policy `P2` *given by:*

```
P2(s) :: if badCredit(s) then deny
```

grants access to individuals whose credit cannot be established. The policy `P2(s) def grant` *grants access to anyone who cannot be proved to have bad credit.*

Coding the BVS policy algebra [11]. We conclude the informal discussion of the untimed combinators in the policy algebra by comparing the expressiveness of our formalism with an existing algebra for composition of policies [11]. The BVS process algebra of policies supports:

1. all boolean operations on policies
2. a scoping operation to enable restriction of a policy to a set of individuals
3. closure of a policy under a set of inference (derivation) rules, where derivation rules are Horn clause logic rules whose head is the authorization to be derived and whose body is the condition under which the authorization can be derived.

In our setting, (1) is explicitly part of the given operations on policies, and (2) is derived from the `if` a `then` P combinator on policies. Our approach to (3) is to encode such rules in the inference relation of the constraint system.

This translation scheme is compositional. That is, the examples analyzed in [11] can be translated compositionally, combinator by combinator into our algebra of policies. We elide these details for reasons of space.

In terms of expressiveness, our work adds defaults and temporal features that are not present in [11].

2.5 The timed language

The temporal constructs in our syntax are powerful enough to encode the temporal behaviors of linear-time temporal logic.

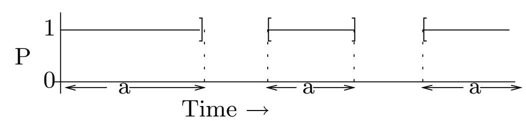
Our operation to add information to the store is `next(tell(`a`))` (for a constraint a). This program imposes the constraint a in the next time instant — the delayed imposition breaks the subtle cycles underlying the tremendous expressive power of defaults. Our programming examples demonstrate that the loss of expressiveness is permissible in this application.

Our basic combinator for timed behavior is `hence` P. Intuitively, a copy of P is created at every time instant, starting at the next instant.

The next two temporal constructs of our process algebra can be viewed by analogy with the scoping operator `if` a `then` P: just as the scoping operator restricts the scope of individuals to whom the policy applies, the temporal operators restrict the “temporal” scope of application of a policy by choosing a subset of time instants at which to execute the policy. This temporal combinator is the “multiform” combinator of synchronous programming. It comes in two variants, both of which provide a subsampled clock to P.

The first is written `time` P `on-present` a In this case, the policy P runs only at the instants when the store entails a, as shown in Figure 8.

Figure 8 Activation diagram for `time` P `on-present` a



Our final temporal combinator is the dual of the above combinator, written `time` P `on-absent` a. In this case, the policy P runs only at the points of time when the store does not entail a.

Defined combinators. Adapting our earlier work [34], given these two basic building blocks and the other untimed combinators, we can program a variety of interesting temporal behaviors:

- A useful variant of `hence` P is `always` $P \stackrel{\text{def}}{=} P$ `left hence` P, which starts a new copy of P at every time instant, including the current time instant.
- The `next(`P`)` combinator that starts P in the next instant.
- P `until` a that behaves like P until a time instant when a is entailed; P is killed from the time instant when a is entailed.
- `first` a `do` P that starts P in the first instant at which a is entailed.

We discuss the laws of the `always` combinator (listed in Figure 9) below. In addition to some laws based on its reading as an LTL operator, the policy combinator `always` also distributes over disjunction and negation. These “extra” equations arise from the fact that policy values are used only at the first instant: recall that at future instants, the continuation does not respond directly to policy queries and its sole role is to constrain the evolution using the shared store. Similar temporal-logic inspired laws hold for the other temporal combinators — we omit them for reasons of space in this extended abstract.

Figure 9 Equational laws for always

$$
\begin{array}{rcl}
\texttt{always}\ (P_1\ \texttt{or}\ P_2) & \equiv & (\texttt{always}\ P_1)\ \texttt{or}\ (\texttt{always}\ P_2) \\
\texttt{always}\ (P_1\ \texttt{and}\ P_2) & \equiv & (\texttt{always}\ P_1)\ \texttt{and}\ (\texttt{always}\ P_2) \\
\texttt{always}\ (\texttt{not}\ P) & \equiv & \texttt{not}\ (\texttt{always}\ P) \\
\texttt{always}\ (P_1\ \texttt{def}\ P_2) & \equiv & (\texttt{always}\ P_1)\ \texttt{def}\ (\texttt{always}\ P_2) \\
\texttt{always next}\ P & \equiv & \texttt{next always}\ P \\
\texttt{always always}\ P & \equiv & \texttt{always}\ P
\end{array}
$$

Coding state machines. From a programming point of view, the primary use of temporal features is to describe state transitions. We begin with a basic example to illustrate the expressiveness of the algebra by a crude and direct encoding of finite state machines.

Example 2.5 *Let $\{s_0, \dots s_n\}$ be the states of a state machine. Let the transition from s_i to s_j be on an a_{ij}. Such a state machine is encoded as follows. Assume a variable s of sort* `State` *with constants $s_0, \dots, s_n$.*

```
time [ always
         [if a00 then next tell (s = s0)
          if a01 then next tell (s = s1)
         ...
         ]
     ] on-present (s = s0)
left
time [ always
         [if a10 then next tell (s = s0)
          if a11 then next tell (s = s1)
         ...
         ]
     ] on-present (s = s1)
```

This demonstrates that the paradigm is expressive enough to address applications exemplified by Example 1.2.

The real utility of the paradigm is that there is no need to construct single monoloithic state machines for the entire policy. We illustrate the implicit compositional construction and use of state machines by considering three classic examples of history-sensitive access control — Chinese Walls [12], dynamic separation of duty [14], and type enforcement in SELinux [26].

The Chinese Wall Security Policy. Brewer and Nash's Chinese Wall security policy [12] is a mandatory access control model that prevents an employee passing data, maliciously or accidentally, from one client to another client when there would be a conflict of interest between the two clients, e.g., when the clients are engaged in legal action but are both represented by lawyers from the same firm. The Chinese Wall model is history-sensitive in the sense that access control decisions are based on the ownership of the data previously read by the user.

In the Chinese Wall policy each object is owned by precisely one company (the Chinese Wall ownership need not be related to the usual notion of ownership for a file object). We write `owner(path)` for the company that owns `path`. Companies with conflicts of interest are encoded using a symmetric binary relation `conf`. We assume the following structure in the constraint system:

Figure 10 Adding A Chinese Wall to the Read Access

```
OKCW(user,path) ::
  (if exists X. [conf(X,owner(path)) /\ read(user,X)]
     then deny)
   def grant

ReadFileAccessWithCW(user,path) ::
   if ReadFileAccess(user,path) then
      [if OKCW(user,path)then
       grant
       left
       always
        (next(tell(read(user,owner(path)))))]
   def deny
```

- A unary function symbol `owner(.)` with sort Path $\rightarrow$ Company, a binary predicate `conf(.,.)` with sort Company $\times$ Company, and a binary predicate `read(.,.)` with sort User $\times$ Path.
- A trivial entailment relation on ground instances. The only other deductions involving these predicates come from the axioms on constraint systems (e.g. $\texttt{conf}(\texttt{o1}, \texttt{o2}) \vdash \exists X.\texttt{conf}(\texttt{o1}, \texttt{X})$).

The simple security component of the Chinese Wall policy is described in Figure 10. To illustrate compositional construction of policies, it is presented as an enhancement of the earlier Windows read access policy. When a read file access is performed, this is recorded in the store for the future. This record is queried by the policy `OKCW` to confirm that no Chinese Wall restrictions are violated. In passing, we also note that the structure of this program permits dynamic changes in the entries of the Chinese Wall conflict table `conf`.

Dynamic separation of duties. We present a minor variant of the Chinese Wall access policy to enforce a form of dynamic separation of duty (DSD) [14].

Technically, DSD relations are described as a pair (role set, n) where n is a natural number greater than 1. The required property is that no user may activate n or more roles from the role set. In this extended abstract, we treat the case $n = 3$. The program in Figure 11 enforces that no three files from the conflict set `conf` are simultaneously being read by a user. In this extended abstract, each conflict set is encoded by a collection of 6 triples, each corresponding to a possible permutation of the 3 elements. The only change from the Chinese Wall program is that the end of access to the file, as recorded by `endaccess(user,path)`, is used to track the reads that are currently being exercised by the user rather than the complete history of accesses as recorded by the policy for the Chinese Wall. The construct P `until` $\texttt{endaccess}(\texttt{user}, \texttt{path})$ naturally fits this intention.

SELinux: RBAC and Type Enforcement. The National Security Agency's Security-Enhanced Linux (SELinux) [26] brings mandatory access control to Linux in the form of role-based access control and type enforcement [10, 3] mech-

Figure 11 Illustrating Dynamic Separation of Duties with Read Access

```
OKDSD(user,path) ::
 (if exists X. exists Y
      [conf(X,Y,owner(path)) /\
       read(user,X) /\ read(user,Y)
      ]
   then deny)
  def grant

ReadFileAccessWithDSD(user,path) ::
  if ReadFileAccess(user,path) then
    (if OKDSD(user,path) then
      grant
      left
      [ always
         next(tell(read(user,owner(path))))
        until endaccess(user,path)
      ])
  def deny
```

anisms to support the tight imposition of the principle of least privilege [27].

Type enforcement associates a *domain* with each process, and a *type* with each resource, such as a file. Types provide a level of indirection for collections of resources that have the same access properties. For example, Kerberos configuration files might be labelled with type `krb5_conf_t` and shell executable files might be labelled with `shell_exec_t`. Access to a resource by a process is primarily determined by the process's domain and the resource's type. For example, the following SELinux policy statement allows a process running in domain `sshd_t`[2] to read a Kerberos configuration file when it is labelled with type `krb5_conf_t`:

```
allow sshd_t krb5_conf_t:file read;
```

Domain transitions are carefully controlled to provide assurance that malicious or compromised software cannot harm the system. For example, the following SELinux policy statement allows a process running in domain `sshd_t` to assume domain `user_t` upon executing a program of type `shell_exec_t`, e.g., `/bin/sh`.

```
domain_auto_trans(sshd_t,shell_exec_t,user_t);
```

In addition to an SELinux user and domain, SELinux associates a role with each process to facilitate the assignment of domains to users. The triple of the user, role, and domain is known as the *security context*. For example, the following SELinux policy statements tell us that the security context (`root`, `system_r`, `sshd_t`) is valid because the `root` user may have role `system_r` and that role may have domain `sshd_t`:

```
user root roles { staff_r system_r };
role system_r types sshd_t;
```

Changes to the user and role in the security context are also restricted because they affect possible values for the domain. The restrictions are imposed using SELinux's constraint language. For example, the following SELinux policy constraint forces the role to stay the same (`r1 == r2`) when a process executes another binary, unless the source domain `t1` is privileged (possesses the `privrole` attribute, e.g., `sshd_t`) and the target domain `t2` is unprivileged (possesses the `userdomain` attribute, e.g., `user_t`):

```
type sshd_t, domain, privrole;
type user_t, domain, userdomain;
constrain process transition
 (r1 == r2 or (t1 == privrole and t2 == userdomain));
```

We formalize a fragment of the SELinux mechanism as a process monitor in Figure 12. The monitor tracks the current security context of the process as a constraint of the form `secCtxt(u,r,d)`. The environment interacts via the input event `ExecTE(u',r',d',type)`, which requests execution of a file of type `type` with a new security context `(u',r',d')`. There is a copy of such code for each tuple of constants `(u,r,d,type)`.

The constraint system determines valid security context transitions using `SELinuxPolicy(u,r,d,u1,r1,d1,type1)`: this is read as permitting a transition from security context `(u,r,d)` to `(u1,r1,d1)` on `type1`. This information is a direct representation of SELinux policy statements. In this extended abstract, we elide the details of the obvious entailment relation of this constraint system.

This program carries forward explicitly the piece of state encoded in `secCtxt(u,r,d)`, using `setState`:

```
setState(u,r,d) :: always(next(tell(secCtxt(u,r,d)))
```

until a change is signalled by another transition. The policy `maintainState`, used above, is a routine case-analysis on the current state:

```
if secCtxt(u1,r1,d1) then
  [setState(u1,r1,d1)
   until exists U,R,D,TYPE. ExecTE(U,R,D,TYPE)
  ]
left
if secCtxt(u2,r2,d2) then
  [setState(u2,r2,d2)
   until exists U,R,D,TYPE. ExecTE(U,R,D,TYPE)
  ]
...
```

Figure 12 A portion of SELinux access control

```
always
  if ExecTE(u1,r1,d1,type1) then
    (if exists U,R,D.
        [secCtxt(U,R,D) /\
         SELinuxPolicy(U,R,D,u1,r1,d1,type1)]
     then [setState(u1,r1,d1)
           until exists U,R,D,TYPE. ExecTE(U,R,D,TYPE)
          ]
     else [maintainState()
           until exists U,R,D,TYPE. ExecTE(U,R,D,TYPE)
          ])
  left
  if ExecTE(u2,r2,d2,type2) then ...
```

[2]The SELinux implementation of type enforcement does not distinguish between domains and types internally, hence the `_t` suffix for domain `sshd_t`.

Figure 13 Operational Semantics

$$\frac{}{u, K \Downarrow K, \mathit{true}, \bot} \qquad \frac{u, P \Downarrow K, v, Q}{u, \mathtt{not}(P) \Downarrow \mathit{not}(K), v, Q} \qquad \frac{u, P_1 \Downarrow K_1, v_1, Q_1 \qquad u, P_2 \Downarrow K_2, v_2, Q_2 \qquad \mathtt{relop} \in \{\mathtt{or}, \mathtt{and}, \mathtt{def}, \mathtt{left}\}}{u, P_1\ \mathtt{relop}\ P_2 \Downarrow \mathit{relop}(K_1, K_2), v_1 \wedge v_2, Q_1\ \mathtt{left}\ Q_2}$$

$$\frac{u \vdash a \qquad u, P \Downarrow K, v, P'}{u, \mathtt{if}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \Downarrow K, v, P'} \qquad \frac{u \not\vdash a \qquad u, Q \Downarrow K, v, Q'}{u, \mathtt{if}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \Downarrow K, v, Q'}$$

$$\frac{u, P \Downarrow \mathtt{true}, v_1, P' \qquad u, Q_1 \Downarrow K, v_2, Q_1'}{u, \mathtt{if}\ P\ \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2 \Downarrow K, v_1 \wedge v_2, P'\ \mathtt{left}\ Q_1'} \qquad \frac{u, P_1 \Downarrow K, v_1, Q_1 \qquad K \neq \mathtt{true} \qquad u, Q_2 \Downarrow K', v_2, Q_2'}{u, \mathtt{if}\ P\ \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2 \Downarrow K', v_1 \wedge v_2, P'\ \mathtt{left}\ Q_2'}$$

$$\frac{u, P[y/x] \Downarrow K, v, Q \qquad y \text{ a fresh variable}}{u, \mathtt{new}\ x\ \mathtt{in}\ P \Downarrow K, v, Q}$$

$$\frac{(u, \vec{y} = \vec{t}), (P[\vec{y}/\vec{x}]\ \mathtt{left}\ (\mathtt{hence}\ \mathtt{tell}(\vec{y} = \vec{t}))) \Downarrow K, v, Q \qquad f(\vec{x}) :: P \qquad \vec{y} \text{ fresh variables}}{u, f(\vec{t}) \Downarrow K, v, Q}$$

$$\frac{u \vdash a \qquad u, P \Downarrow K, v, Q}{u, \mathtt{time}\ P\ \mathtt{on\text{-}present}\ a \Downarrow K, v, \mathtt{time}\ Q\ \mathtt{on\text{-}present}\ a} \qquad \frac{u \not\vdash a}{u, \mathtt{time}\ P\ \mathtt{on\text{-}present}\ a \Downarrow \bot, \mathit{true}, \mathtt{time}\ P\ \mathtt{on\text{-}present}\ a}$$

$$\frac{u \not\vdash a \qquad u, P \Downarrow K, v, Q}{u, \mathtt{time}\ P\ \mathtt{on\text{-}absent}\ a \Downarrow K, v, \mathtt{time}\ Q\ \mathtt{on\text{-}absent}\ a} \qquad \frac{u \vdash a}{u, \mathtt{time}\ P\ \mathtt{on\text{-}absent}\ a \Downarrow \bot, \mathit{true}, \mathtt{time}\ P\ \mathtt{on\text{-}absent}\ a}$$

$$\frac{}{u, \mathtt{hence}\ P \Downarrow \bot, \mathit{true}, P\ \mathtt{left}\ (\mathtt{hence}\ P)} \qquad \frac{}{u, \mathtt{next}(\mathtt{tell}(a)) \Downarrow \bot, a, \bot}$$

3. SEMANTICS

3.1 Operational semantics

The operational semantics of the policy algebra is presented in Figure 13.

It is assumed that the security monitor possesses an initial store for the system. The security monitor invokes a policy with appropriate terms (respecting arity and sort), e.g.:

```
ReadFileAccess(Alice,"\Patients\Chart01.txt")
```

or forwards input events of the kind `endaccess`(s, o, r). Evaluation may fail to terminate because of recursive calls. If evaluation of a process in a store terminates, then the resulting value K is one of `true`, `false`, or $\bot$.

The operational semantics judgement is

$$u, P \Downarrow K, v, Q$$

A configuration (u, P), consisting of a store u and a process P (that may have free variables), is evaluated to a value K and a continuation configuration (v, Q) for the next time instant. In the course of evaluation, P can query the current instant's store u, but the store u is never modified significantly in the current instant. When the security monitor handles the next access request, say $f(\vec{t})$, by invoking a policy at the next logical time instant, that policy runs in parallel with the continuation store and process from the previous time instant, i.e., the program executed at the next time instant is $f(\vec{t})$ `left` Q in the store v.

Note on new variable creation. The operational rules in Figure 13 assume new variable creation in the rules for procedures and new: we assume that these new variables are drawn from a global shared namespace, so each request for a new variable yields a globally distinct new identifier.

3.2 Equational reasoning

In this section, we define a notion of behavioral equivalence between policies. Our definitions are inspired by similar studies for the timed cc languages [29]. However, our approach is tuned to the current setup and is intentionally more syntactic to enable coinductive proofs directly on policies as they execute.

We consider a simple modification to the operational semantics relation:

$$\mathcal{X}, u, P \Downarrow K, \mathcal{Y}, u', P'$$

where $\mathcal{X}$ is a set of variables and (u, P) is a configuration. The idea is that the new variables created in operational reductions from the configurations (u, P) are not from $\mathcal{X}$. Figure 14 gives the flavor of the modified rules.

We now define a notion of behavioral equivalence between configurations and then processes. Consider a ternary relation $\mathcal{R}$ with members of the form $(\mathcal{X}, (u, P), (v, Q))$, where $\mathcal{X}$ is a set of variables and (u, P), (v, Q) are configurations. When $\mathcal{R}$ is bisimilarity (defined presently), the meaning is that (a) the stores u, v are identical when the $\mathcal{X}$ vars are hidden, and (b) (u, P) and (v, Q) have the same behavior when executed with any "input" constraint that does not meaningfully involve variables in $\mathcal{X}$.

Figure 14 Excerpt of Extended Operational Semantics

$$\frac{}{\mathcal{X}, u, K \Downarrow K, \mathcal{X}, true, \bot} \qquad \frac{\mathcal{X}, u, P \Downarrow K, \mathcal{Y}, v, Q}{\mathcal{X}, u, \mathtt{not}(P) \Downarrow not(K), \mathcal{Y}, v, Q}$$

$$\frac{\mathcal{X}, u, P_1 \Downarrow K_1, \mathcal{Y}, v_1, Q_1 \qquad \mathcal{X}, u, P_2 \Downarrow K_2, \mathcal{Z}, v_2, Q_2}{\mathcal{X}, u, P_1 \ \mathtt{left}\ P_2 \Downarrow K_1, \mathcal{Y} \cup \mathcal{Z}, v_1 \wedge v_2, Q_1 \ \mathtt{left}\ Q_2}$$

$$\frac{\mathcal{X} \cup \{y\}, u, P[y/x] \Downarrow K, \mathcal{Y}, v, Q}{\mathcal{X}, u, \mathtt{new}\ x\ \mathtt{in}\ P \Downarrow K, \mathcal{Y}, v, Q}$$

$$\frac{f(\vec{x}) :: P \qquad R = (P[\vec{y}/\vec{x}]\ \mathtt{left}\ (\mathtt{hence}\ \mathtt{tell}(\vec{y} = \vec{t}))) \quad \mathcal{X} \cup \{\vec{y}\}, (u, \vec{y} = \vec{t}), R \Downarrow K, \mathcal{Y}, v, Q}{\mathcal{X}, u, f(\vec{t}) \Downarrow K, \mathcal{Y}, v, Q}$$

The ternary relation $\mathcal{F}(\mathcal{R})$ is defined by:

$$\begin{array}{l} (\mathcal{X}, (u, P), (v, Q)) \in \mathcal{F}(\mathcal{R}) \Leftrightarrow \\ \quad \exists \mathcal{X}.u = \exists \mathcal{X}.v, \text{ and} \\ \quad \forall a, K. \\ \qquad \forall \mathcal{Y}, u', P'.\ \ \mathcal{X}, (u \wedge \exists \mathcal{X}.a), P \Downarrow K, \mathcal{Y}, u', P' \Rightarrow \\ \qquad \exists \mathcal{Z}, v', Q'.\ \ \mathcal{X}, (v \wedge \exists \mathcal{X}.a), Q \Downarrow K, \mathcal{Z}, v', Q' \\ \qquad\qquad \wedge (\mathcal{Y} \cup \mathcal{Z}, (u', P'), (v', Q')) \in \mathcal{R} \end{array}$$

The functional $\mathcal{F}$ is monotone. A bisimulation is a relation $\mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$ and $\mathcal{R}^{\mathrm{op}} \subseteq \mathcal{F}(\mathcal{R}^{\mathrm{op}})$. Bisimilarity $\sim$ is defined to be the largest bisimulation.

Definition 3.1 *P and Q are bisimilar, written $P \equiv Q$, when $(\emptyset, (u, P), (u, Q)) \in \sim$, for all stores u.*

Coinduction provides good proof-rules for bisimilarity. The following theorem testifies that we also have a compositional proof principle.

Theorem 3.2 *Bisimilarity is a congruence.*

The proof of this theorem formalizes the idea that our definitions capture the observable elements of program behavior, namely the shared store and the result to the policy queries. The proofs involving the `new` combinator illustrate the name-management techniques supported by our definitions. The equational laws that were described in earlier sections are proved using the usual bisimulation techniques.

3.3 Analysis

In the rest of this section, we assume that we are working with finite state policies: this restricts the scope of the analysis to systems with a finite number of principals, resources, roles, and rules. We develop a framework to perform security analysis with dynamic state-dependent restrictions — [25] leaves this as an open problem.

We consider the labelled transition system whose states are configurations (u, P) and whose transitions are given by input events and policy invocations. For example, consider the following transitions:

- $(u, P) \xrightarrow{f(\vec{t}), K} (v, Q)$ if $u, (f(\vec{t})\ \mathtt{left}\ P) \Downarrow K, v, Q$
- $(u, P) \xrightarrow{\mathtt{endaccess}(s,o,r)} (v, Q)$ if $u \wedge\ \mathtt{endaccess}(s, o, r), P \Downarrow K, v, Q$

This labelled transition system is clearly similar to the models in state-transition approaches to trust-management [13]. What we have provided via a policy algebra is a structured approach to building this LTS. In our examples, each state of the LTS is captured by a Datalog constraint program.

Next, we show that the properties considered in [23] are expressible in LTL. In this extended abstract, we merely recall that an LTL formula is interpreted over traces (sequences of states). In each state of the trace, a truth value is associated with each of the atoms appearing in the formula. LTL has temporal operators in addition to the usual logical connectives of propositional logic so that one can describe relationships between the values of the atoms across time. The formula $\mathbf{G}[\ f\ ]$ is true in a state of the trace if the subformula f is true from then on. The formula $\mathbf{F}[\ f\ ]$ is true in a state of the trace if the subformula f is true in that state or in some future state. $\mathbf{X}[\ f\ ]$ is true in a state of the trace if f is true in the next state in the trace. LTL has also been extended in order to talk about past events by defining past time versions of each of these operators.

When the LTL formula is used to specify the behavior of a system, there is an implicit quantification performed over all traces that begin in the start state of the system, i.e., a system satisfies the LTL formula if every trace satisfies that formula.

For convenience in writing these specifications, we assume the following shortcuts. The predicate $\mathrm{grant}(p, r)$ is true at the moment that p is granted access to r and is false otherwise. The predicate $\mathrm{endaccess}(p, r)$ is true at the moment that p releases resource r and is false otherwise. The predicate $\mathrm{tryaccess}(p, r)$ is true if a request by p for resource r would be granted and is false otherwise. This special predicate is required when checking for availability since we cannot force a principal to actually make the request.

We also assume the availability of some domain-specific predicates like $\mathrm{safe}(p)$, $\mathrm{prop}_1(p)$, $\mathrm{prop}_2(p)$ as well as predicates like $\mathrm{admin}(p)$ whose truth values can be deduced from the store/state using the techniques of [23].

In writing these properties, we freely use quantification. Since the number of principals is finite, quantification over principals is achieved via finite conjunction.

We first specify the properties considered in [25] as LTL safety properties[3], as revealed by their description using the G-operator ("true along the entire path").

Simple Safety Is a (presumably untested) principal guaranteed to be denied access to a resource?

$$\mathbf{G}[\ \neg \mathrm{grant}(p, r)\ ]$$

Simple Availability Is a (presumably trusted) principal guaranteed access to a resource?

$$\mathbf{G}[\ \mathrm{tryaccess}(p, r)\ ]$$

Bounded Safety Is the set of principals having access bounded by a given set of principals?

$$\forall p\ .\ \mathbf{G}[\ \mathrm{grant}(p, r) \rightarrow \mathrm{safe}(p)\ ]$$

[3]There is a clash of vocabulary between the use of "safety" in [25] and the use of the same word in LTL. We explicitly say LTL safety.

Containment Does every principal that has one property have another property?

$$\forall p \ . \ \mathbf{G}[\ \mathrm{prop}_1(p) \rightarrow \mathrm{prop}_2(p)\ ]$$

The algorithms to verify these properties for finite-state systems are evident: standard LTL techniques are used to explore the state space and Datalog techniques as explored in [25] establish properties at individual states.

The advantage of viewing the analysis as an instance of extended LTL model-checking is that we are able to perform security analysis with dynamic state-dependent restrictions. Clearly, the Chinese Wall specification is of this nature:

$$\forall p.\mathbf{G}[\ \mathrm{grant}(p, r_1) \rightarrow \mathbf{G}[\ \neg\mathrm{grant}(p, r_2)\ ]\ ]$$

This formula captures the specification that when a given principal P is given access to a resource r_1, she can no longer access another resource r_2.

Such dynamic assertions are particularly helpful in compositional analysis of policies, as illustrated by the following example.

- Policy P_1 enforces the Chinese Wall policy between resources r_1, r_2 as specified above.
- Process P_2 controls who gets access to r. Let us assume that it only allows principals who have been granted access to r_2 in the past (coded as $\mathbf{F}_{\mathsf{past}}[\ \mathrm{grant}(p, r_2)\ ]$) to gain access to r:

$$P_2 \models \forall p.\mathbf{G}[\ \mathrm{grant}(p, r) \rightarrow \ \mathbf{F}_{\mathsf{past}}[\ \mathrm{grant}(p, r_2)\ ]\ ]$$

The conjunction of the two policies ensures that if a principal p accesses r_1, then she does not have access to r.

4. CONCLUSIONS

The requirements of precise formal foundations and emphasis on avoidance of failure make the general area of security particularly receptive to the application of declarative programming techniques. This observation is of course not novel, given the variety of literature referred to in the introduction.

The novelty in this paper is the application of declarative techniques from reactive programming languages. Broadly speaking, deterministic reactive programming is particularly appropriate for security applications for two reasons. Firstly, it supports precise and expressive executable specifications (as evidenced by their use in embedded systems). Secondly, its foundations provide a framework that supports the incorporation of powerful analysis methods (as evidenced by the success of software model-checking for reactive systems).

This paper is a first step in this exploration. We have demonstrated a policy algebra that attempts to modularly combine untimed constraint-based logical reasoning with temporal reasoning. This view enables us to build on and generalize existing research to history-sensitive policies.

Foundationally speaking, we feel that our methods provide a general recipe to add temporal features to existing logical formalisms. We intend our ongoing and future work on implementations to provide pragmatic justification.

5. REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. Network and Distributed System Security Symp.*, 2003.

[2] M. Backes, M. Dürmuth, and R. Steinwandt. An algebra for composing enterprise privacy policies. In P. Samarati, D. Gollmann, and R. Molva, editors, *ESORICS*, volume 3193 of *Lecture Notes in Computer Science*, pages 33–52. Springer, 2004.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.

[4] S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 190–199. ACM Press, 2004.

[5] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.

[6] A. Barth, J. C. Mitchell, and J. Rosenstein. Conflict and combination in privacy policy languages. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 45–46. ACM Press, 2004.

[7] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11 – 17. Elsevier Science Publishers B.V. (North Holland), 1989.

[8] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[9] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. IEEE Conf. Security and Privacy*. IEEE Press, 1996.

[10] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[11] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[12] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, 1989.

[13] A. Chander, D. Dean, and J. C. Mitchell. Reconstructing trust management. *Journal of Computer Security*, 12(1):131–164, 2004.

[14] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Information System Security*, 4(3):224–274, 2001.

[15] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[16] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *CSFW '03: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 118–130. IEEE Computer Society, 2003.

[17] J. Y. Halpern and V. Weissman. A formal foundation for XrML. In *CSFW '04: Proceedings of the 17th*

IEEE Computer Security Foundations Workshop (CSFW'04), pages 251–263. IEEE Computer Society, 2004.

[18] D. Harel and A. Pnueli. *Logics and Models of Concurrent Systems*, volume 13, chapter On the development of reactive systems, pages 471–498. NATO Advanced Study Institute, 1985.

[19] P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.

[20] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[21] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[22] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[23] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 58–73. Springer-Verlag, 2003.

[24] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 2002.

[25] N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *IEEE Symposium on Security and Privacy*, pages 123–139. IEEE Computer Society, 2003.

[26] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[27] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[28] M. McDougall, R. Alur, and C. A. Gunter. A model-based approach to integrating security policies for embedded devices. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 211–219. ACM Press, 2004.

[29] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.

[30] J. Park. *Usage control: a unified framework for next generation access control*. PhD thesis, 2003.

[31] J. Park and R. S. Sandhu. The $\mathrm{UCON}_{\mathrm{ABC}}$ usage control model. *ACM Trans. Information System Security*, 7(1):128–174, February 2004.

[32] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.

[33] V. A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

[34] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, November/December 1996.

[35] V. A. Saraswat, R. Jagadeesan, and V. Gupta. jcc: Integrating timed default concurrent constraint programming into Java. In F. Moura-Pires and S. Abreu, editors, *EPIA*, volume 2902 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2003.

[36] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, pages 333–352, January 1991.

[37] F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *FMSE '03: Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, pages 32–42. ACM Press, 2003.

[38] E. G. Sirer and K. Wang. An access control language for web services. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 23–30. ACM Press, 2002.

[39] M. M. Swift, P. Brundrett, C. Van Dyke, P. Garg, A. Hopkins, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control for windows 2000. *ACM Transactions on Information and System Security*, 5(4), Nov 2002.

[40] V. N. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *NSPW '02: Proceedings of the 2002 Workshop on New Security Paradigms*, pages 61–68. ACM Press, 2002.

[41] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 45–55. ACM Press, 2004.

[42] D. Wijesekera and S. Jajodia. Policy algebras for access control — the predicate case. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 171–180. ACM Press, 2002.

[43] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, 2003.

[44] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 1–10. ACM Press, 2004.