A Framework for Decentralized Access Control

Meenakshi Balasubramanian[*] Abhishek Bhatnagar
Namit Chaturvedi Atish Datta Chowdhury Arul Ganesh
Research and Technology Group
Honeywell Technology Solutions
Bangalore - 560076
India

ABSTRACT

We present a framework for decentralized authorization for physical access control, using smart cards, where access to individual rooms is guarded by context-dependent policies that are dynamically evaluated. Policies are specified using a logical language parameterized by events. A policy analyzer converts policy specifications into equivalent executable automata and also generates initialization information about the contexts used in these policies. While the automata are stored in users' smart cards, context initialization information is disseminated in the system. We also provide a context modeling mechanism that supports construction and propagation of contexts in the system. Upon an access request, user automata are executed at the point of access in the presence of current context information. This results in an allow/deny decision. The benefit of this approach lies in resolving authorizations in a decentralized manner in situations where the solution needs to scale with increasing number of users.

Categories and Subject Descriptors

J.7 [**Computer Applications**]: Computers in Other Systems—*Embedded controllers*; F.4.3 [**Theory of Computation**]: Formal Languages—*Classes defined by grammars or automata*; F.4.1 [**Mathematical Logic**]: Model Theory; C.3 [**Special-Purpose and Application-Based Systems**]: Smartcards

General Terms

Design, Security

Keywords

Authorization, decentralization, physical access control

[*]Author names appear in alphabetical order. All email addresses are FirstName.LastName@honeywell.com

1. INTRODUCTION

The domain of access control involves solutions to the problems of authorization, validation, and authentication. The goal of authorization is to specify and evaluate a set of policies that control the access of users to resources, resulting in grant or denial of access. Validation usually refers to securely verifying the authorized privileges, and the goal of authentication is to prove the identity that a user claims. In this paper, we present an framework for decentralized authorization for physical access control, using smart cards, where access to resources is guarded by context-dependent policies that are dynamically evaluated. In this context, resources are rooms, which can be defined as enclosures guarded by entrances or doors. In traditional implementations[1] of physical access control, the doors are equipped with card readers, which are either connected to a central controller, or a locally cached image of the central controller. When a user presents the access card to the reader, the latter communicates the card information to the controller and waits for a reply instructing whether or not to allow access.

In these traditional systems, access cards and card readers are passive devices without any processing power of their own. The central controller, on the other hand, is a well designed sophisticated device with fail-over capabilities, and advanced hardware and algorithms to enable fast decision making. Policies dictating access are predominantly specified using Access Control Lists (ACLs) that describe static policies, e.g. "user X is not allowed in room R". The decision making process of the central controller includes look-up of these lists to determine whether or not a user has privileges to enter a room.

These static policies are in contrast with context dependent policies where the conditions determining access change dynamically, and must be evaluated at the time of request. For example, a policy may prohibit a user from entering into the lobby if all the rooms accessible from the lobby have reached their maximum limits. In such a situation, a central controller will have to collate information from each of the rooms before making a decision about access to the lobby. It is envisioned that physical access control in buildings, facilities, and townships of the future will have a significant reliance on context dependent policies. Therefore, the central controller will be burdened with information from all over the facility, and each piece of information may possibly

[1]As exemplified in several current commercial physical access control solutions.

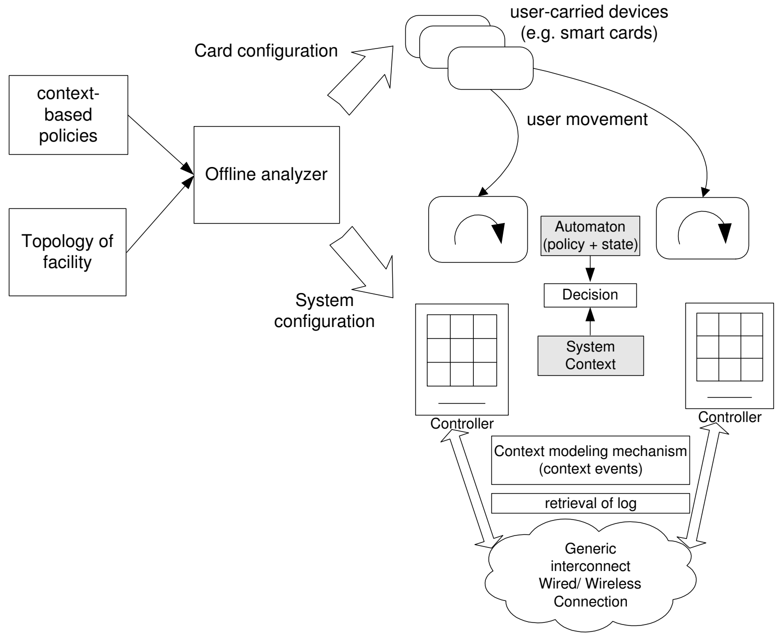

Figure 1: Decentralized access control framework: Overview

contribute toward constructing many kinds of contexts for different users. In the centralized systems of today, particular solutions enabling dynamic context handling are built in as instances of specific programs or triggers on the central controller, and such solutions will not scale with increasing number of users in the facility.

State of the art physical access control solutions are moving toward using existing general purpose building networks for communications between various card readers and the central controller. Reasons of flexibility and ease of installation will keep driving this trend. A centralized solution for context dependent authorization does not suit well with a generic interconnect. This unsuitability is due to the inherent dependency on the central controller for every decision.

In this paper we present a decentralized framework for physical access control based on dynamic authorization that addresses the issues mentioned above. In our framework, we replace the central controller with a network of smaller controllers that maintain system information (context) and introduce per-user active devices with processing power and memory. Smart cards, Java enabled smart buttons, SIM cards fit precisely in our definition of small, portable, and intelligent user devices. Policies are specified in a formal language, and are analyzed and finally stored as executable automata in smart cards carried by users. Context information is dynamically maintained amongst the controllers using a context modeling mechanism. Upon access request a controller and the smart card come together to execute the policies stored in the user's card, and decide whether to grant or deny access based on the context information.

We provide an overview of our framework for decentralized access control in the next section. Section 3 provides a study of related works. Section 4 gives details on the formal logical language used to specify the policies, a model for their execution, and our context modeling mechanism. Section 5 summarizes our approach toward realizing this framework. We then look at the benefits of our approach, followed by our conclusion and identification of areas for future efforts.

2. DECENTRALIZED ACCESS CONTROL FRAMEWORK - OVERVIEW

The proposed system framework for physical access control is shown in Figure 1. We refer to it as *Decentralized Access Control* (DAC) framework. The components of this framework are:

- User-carried devices: These are active devices with built in computational capabilities and memories. Every user carries one such device and uses it to make access requests. In this paper we use smart cards [9] as representatives of these devices.
- Room: A room is an enclosed space in a facility, access to which is obtained through doors.
- Door: A door refers to an element from the set of physical entry/exit points to rooms, which are access control enabled, and hence act as access agents for rooms. Each door has a reader on either side, and an actuator responsible for opening the door.
- Reader: This is a device installed on a door, which can read from and write to the smart cards. Users request access by either bringing their smart cards in close vicinity of the readers, or by swiping or inserting them in appropriate reader slots.
- Controller: Readers of each door are connected to microcontroller based devices, called controllers. Readers from many doors may be connected to the same controller. A controller is expected to be connected over a network with other controllers, and have reasonable

processing capabilities and memory. The functionality of a reader and a controller may also be incorporated into one integrated unit. In such a case, a door will have a unit each at both of its sides.

- Interconnect: Controllers are connected to each other through a network installation that is referred to as the interconnect, e.g. the IP network of a facility. It may include wired or wireless communication components. Devices like special purpose servers, required for log keeping etc, are also connected with the interconnect.

The smart cards carry information about all access privileges of the corresponding user. The controllers, in collaboration or isolation, maintain information about the system context. Upon an access request, a decision is taken locally by the virtue of interaction of smart card and a controller. The interconnect is used to transfer system-level context information among pertinent set of controllers.

The topology of the facility is described in terms of rooms and their neighborhood relationships. Consider the example facility in Figure 2, which comprises of five rooms, viz. A, B, C, D, and outside of the facility - W. The topology of this facility is given in Table 1. Each door in the facility has readers on either sides that are connected to controllers. By the virtue of these connections a controller gets associated with the rooms on either side of the doors it is connected with. The controller then becomes a participant in maintaining context information for these rooms. For now we assume that all of the controllers are connected with each other over a network. In this figure we show that both the readers of every door are connected to a controller, however in practice these connections can be arbitrary with multiple readers connected to one controller.

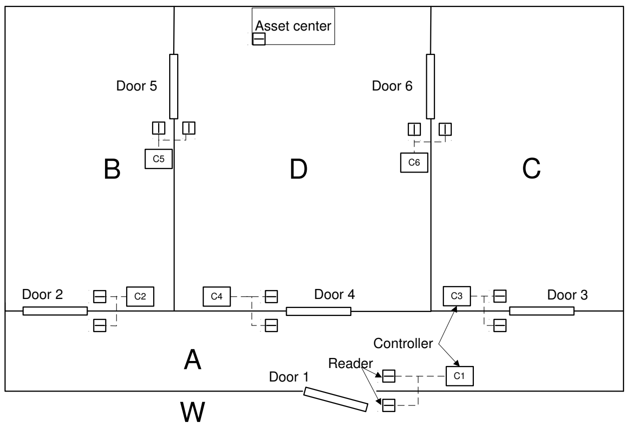

Figure 2: Example facility layout

In addition to the door controllers, the facility may have other controllers, PCs or devices that participate in generating system context pertinent to various users. For example, Room D has an asset center that contributes an "asset-issued" and an "asset-returned" context to smart cards every time the user issues or returns the asset, respectively.

A facility administrator divides all the users into different classes or roles, and defines privileges for each of these roles. This requires defining the contexts pertinent to various roles. These contexts are then used to construct access

```
# Input topology - List of rooms
rooms :  A,B,C,D,W;
```
```
# Input topology - Neighborhood
neighbor A: C,B,D,W;
neighbor B: A,D;
neighbor C: A,D;
neighbor D: A,B,C;
neighbor W: A;
```

Table 1: Facility topology specification

policies corresponding to each of these roles. Contexts are defined in terms of events. An *event* is an occurrence or a happening of significance to one or more policies, and is represented in our framework as an identifier label. The set of events includes *user actions*, *application actions*, and *context events*. User actions include user behaviors like request for services; application actions include grant/denial of services; context events are constructed with the help of application actions and/or user actions and/or other context events. For every event representing some context, there is a *dual event* that represents the absence of the same context. Hence, for context events, either the event or its dual is always true. Henceforth, we use the terms *context* and *context event* interchangeably. In our notation, the dual of a context Z is represented by Z^d. *Behavior* of the system is modeled as a sequence of events, in the order in which they occur when the access control application is being executed.

Policies are specified in a formal logical language parameterized by events. The offline analyzer module, as shown in Figure 1, checks for inconsistencies amongst policies, and converts the specification to an executable format. We use conventional finite state automaton [17] as the executable format of choice. The automaton for each role is stored in the smart card of the user belonging to that role. Smart cards also act as a storage for events specific to an individual. For example, user actions like issuance and return of assets can be recorded on the user's card as *user history*.

Given the facility topology, event definitions and the policies that use these events, the offline analyzer also creates configuration information for each of the controllers for a) detecting and/or composing the events, and b) communicating them to other controllers. This is done using a *context modeling mechanism*. The controllers are programmed with this information, which enables them to co-operate in maintaining local system contexts. This mechanism ensures that the context information in the system is up-to-date. Whenever user cards are presented at a reader connected with a controller, this mechanism also instructs the controller to provide the cards with context events that are pertinent to the user-role. Policies enforce access depending on these events. For example, we could have a policy which states that a requesting user may only access the room if an escort has entered it through the same door no longer than 7 seconds ago. In such a case, time and point of entry of the escort contribute to composition of an event that decides whether the user will be allowed or denied.

The controllers can buffer and forward their local audit trail of request and responses at a central server for log keeping. This communication may be given a lower priority as compared to that for event exchange, hence it will not put any undue constraint on maintaining the context.

We would like to emphasize here that the aim of our framework is to support policies whose evaluations depend upon dynamically varying context. Policies themselves are not dynamic. Users may be expected to re-program their cards at an agreed upon granularity so that they can reflect any change in policies. However, this exercise will only be required if the changes pertain to user specific history context. This is because individual user history is maintained on user cards. By virtue of segregating user and system contexts, we can maintain system contexts within the network of controllers. System context definitions may be changed at any time to reflect a change in policy intent without disturbing the cards.

3. RELATED WORK

Bauer et al [4] also use a formal logical language to specify access control policies and present a distributed algorithm that assembles a proof from various pieces of proofs provided by entities involved in their physical access control system. Each user carried "smart phone" is equipped with a theorem prover, which assists in distributed proof generation for authorization decisions. The distributed proof generation algorithm is guaranteed to terminate whenever a centralized prover system also terminates. Our system, on the other hand, maintains system context in a collaborative manner, and authorization is done locally. The authorization mechanism is based on finite state machines, and is always guaranteed to terminate in constant time.

Abadi et al [2] work with a formal modal logic to specify access control policies. While the language is tuned toward specifying distributed policies, the paper does not provide a decentralized execution mechanism. The ADAM architecture [24] presents a distributed execution mechanism defining user agents and authorization agents and highlights when and how a response to an access request is arrived at. However, their architecture does not focus on a formal model to specify or execute policies.

Abadi presents a partial survey of logic based policy specification and evaluation in access control [1]. Almost all the works cited therein follow proof theoretic approaches to reason about authorization. A comprehensive survey of the various access control policies, models and mechanisms is also provided by Samarati and di Vimercati [23]. Bertino et al [6] emphasize the need for dynamic authorizations in a collaborative environment. They use workflow management system as an example to underline the need for a formal model to specify constraints on policies. This includes dynamic constraints i.e. those that cannot be verified from the definition of the workflow, other than at runtime. Jajodia et al [15] extend this approach in a more generic and flexible way. They introduce a formal mechanism of expressing authorization specifications which is rich enough to incorporate any application-specific requirements within the fold of the authorization framework. In another work, Bertino et al [5] provide a language and a model to capture dynamism in terms of time. We can represent time as context in our framework.

These approaches concentrate more on access control as modeled on computer systems in general and not on physical access control in buildings in particular. Consequently, their focus is on languages that provide flexibility in specifying role based policies, and on guaranteeing unambiguous evaluation (decision) with feasible bounds on the run time and not on decentralized implementation. The functional architecture of many of these approaches assumes a centralized authorization resolver that would decide on a grant/denial, given an access request, a history of the system and the authorization specification policies. We develop a new policy language based on MSO logic, which gives us three advantages. First is that MSO provides a formal groundwork to support powerful policies, and can uniformly accommodate the changing nature of policy requirements. Secondly, MSO representation make it easy to translate policies into finite state automata for deployment on small devices. And lastly, many previous formal approaches exploit proof theoretic mechanisms. Subsequently, a theorem proving algorithm is not guaranteed to terminate whereas an automaton based policy evaluation terminates in constant time.

Context dependent physical access control can be viewed as a special case of a wider class of ubiquitous computing applications, which are deployed in smart spaces. Such applications are characterized by interactions of smart devices [22, 8]. These devices are classified as *clients* and *sentries*, referring to an arrangement where users carry small, intelligent, and mobile client devices that seek certain services from a network of stationary sentry nodes [3]. Their interactions are governed by continuously evolving contexts.

The Context Toolkit by Dey et al [10] is one of the initial works in formalizing context and its use in context-aware applications. Dey et al present a conceptual framework for prototyping of context-aware applications that provides considerable understanding of functional abstractions toward developing a software architecture. They provide a foundation for a general purpose support for various context dependent applications running in tandem, and possibly sharing resources. Our approach, however, is tuned to the domain of physical access control.

Ranganathan and Campbell's first order logic based context modeling [21] deploys ontology based specification of context predicates such that "it allows different components in the system to have a common understanding of the semantics of different contexts." We also incorporate logic programming techniques to reason about contexts tuned to our application. In our framework, the application behavior is described in terms of pre-defined context events. Each event is owned by a unique controller, which hosts the logic program that reasons about the context. The semantics of an (application, user, or context) event is useful and meaningful only to the owner of the event. Other controllers can only subscribe to the values of these events, which may be True or False signifying the occurrence or absence of the corresponding event.

4. POLICY SPECIFICATION LANGUAGE, EXECUTION MODEL, AND CONTEXT MODELING

The Decentralized Access Control (DAC) framework includes a language to define complex policies with features to handle various parameters of dynamism, like user's history of movement, context induced by events in various rooms of the facility etc. Typical examples of context-based policies are:

- Room count: Certain users cannot enter a room if the number of users reaches a certain value.

- Connected count: If the number of users in a set of rooms have collectively reached a stipulated number, access to certain other rooms or specific areas (e.g. lobby) should be denied, thereby ensuring regulated movement.
- Supervisor required: A user may enter the room only if a supervisor or a security guard is already present.
- Escorted access: A user may enter a room (through a door) only if a designated escort has entered the room (through the same door) no longer than, say 7 seconds ago.
- Interlocking of doors: Certain rooms can only be accessed if a surrounding protective space is first closed.
- Anti-passback: Users who don't have a record of making a legal exit from a room will be denied next time they try to access the same and/or any other room in the facility.
- Guard tour: At specified intervals during the days, a guard is required to tour a facility. He/she must visit the rooms in a specific order, and must not spend any longer than specified amounts of time in each.

4.1 Formal logical language

We use *Monadic Second Order (MSO) Logic* [26] parameterized by events of the system as the formal language for policies. Syntax of the logic is built over a set of first and second order variables, which are used to quantify over events of the system. First order variables quantify over individual events, while second order variables are used to quantify over a set of events. We represent a first order variable in lower case, e.g. x, and a second order variable in upper case, e.g. X. Policies are formulae of the logic, and are built on top of certain elementary relations over these first and second order variables using Boolean operations and quantification. Relations talk about how a variable represents a particular event and about the order of occurrence of events.

4.1.1 Syntax

The set of events is denoted by Σ, and may include:

1. Labels for the user events such as:
   - *request-entry-A*, *allow-entry-A*, enumerated for every room A specified in the topology.
2. Labels for context events, as defined by the administrator, of the form:
   - A_{max} representing the event corresponding to the occupancy of room A reaching its maximum allowable value.
   - A_{super} representing the event that a supervisor is present in room A.
   - A_{closed} representing the event that all doors of room A are closed.

Note that the set of context events will be exhaustively defined by the administrator to represent all the conditions required by the policies. For each of the context events, their duals will also get defined, referring to the events representing the absence of the respective events. E.g A^d_{max} is the dual event of A_{max}, and represents the event corresponding to occupancy in room A being less than the maximum.

Events of the system are used to construct *atomic formulae*. The atomic formulae are given by:

- For each event $e \in \Sigma$, we have a predicate $e(x)$ which represents the fact that the label of the event represented by the variable x is e.
- For the first order variables x, y the predicate $x \leq y$ represents the fact that the event corresponding to y occurs after the event corresponding to x or x refers to the same event as y in a computation of the system.
- For first order variables x, y, the predicate $x < y$ represents the fact that the event corresponding to y occurs immediately after the event corresponding to x in a computation of the system.
- For a first order variable x and a second order variable X, the atomic formula $x \in X$ represents the fact that the event corresponding to the variable x belongs to the set of events corresponding to X.

Formulae depicting policies are built from the atomic formulae using the following connectives:

- Boolean operators representing *negation* and *disjunction* are $\neg$ and $\vee$ respectively. The operators for *conjunction* ($\wedge$), *implication* ($\Rightarrow$), and *equivalence* ($\equiv$) can be derived from negation and disjunction. Note that $<$ predicate above can be expressed using $\leq$ and $\neg$.
- The operators *for all* ($\forall$) and *there exists* ($\exists$) will be used to quantify over first and second order variables.

To summarize, the syntax of the policy language is MSO logic tuned for physical access control. Each access control policy is defined as a formula using the above syntax.

4.1.2 Semantics

Semantics of policies will be defined using *words* over the alphabet Σ. Words are finite sequences of events from Σ that represent behaviors. Consider a formula φ. φ is interpreted over a word w as follows: An *interpretation* of first and second order variables is a function I that assigns a letter of Σ to each first order variable and a finite set of letters of Σ to each second order variable. These letters occur as labels of positions in a word when a formula (policy) is interpreted over it. For a formula φ, let V_φ denote the set of variables that occur free in φ, i.e. they are not in the scope of any quantifier in φ. Interpretation is then nothing but a function $I : V_\varphi \to \Sigma$.

The notion of when a word w *satisfies* a formula φ, under an interpretation I is given by $w \models_I \varphi$ and is defined inductively as follows:

- $w \models_I e(x)$ iff $I(x) = e$.
- $w \models_I x \leq y$ iff either $I(x) = I(y)$ or $I(x)$ occurs before $I(y)$ in the word w.
- $w \models_I x \in X$ iff $I(x) \in I(X)$.
- $w \models_I \neg\varphi$ iff it is not the case that $w \models_I \varphi$.
- $w \models_I \varphi_1 \vee \varphi_2$ iff $w \models_I \varphi_1$ or $w \models_I \varphi_2$.

$$
\begin{array}{l}
\forall x, \forall y \ (x \leq y \ \wedge \ C^d_{max}(x) \wedge \textit{request-entry-C}(y) \ \wedge \ \neg\exists z \ (x \leq z \ \wedge \ z \leq y \ \wedge \ C_{max}(z)) \\
\qquad \Rightarrow \\
\qquad \exists w \ (y < w \wedge \textit{allow-entry-C}(w)) \\
\qquad) \\
\wedge \\
\forall w \ (\textit{allow-entry-C}(w) \\
\quad \Rightarrow \\
\quad \exists x, \exists y \ (x \leq y \ \wedge \ C^d_{max}(x) \wedge \textit{request-entry-C}(y) \wedge \ \neg\exists z \ (x \leq z \ \wedge \ z \leq y \ \wedge \ C_{max}(z)) \\
\qquad\qquad \wedge \ y < w \\
\qquad\qquad) \\
\quad)
\end{array}
$$

Figure 3: Example policy formula

- $w \models_I \exists x \ \varphi$ iff there exists an interpretation function I' that extends I by assigning an event to the variable x such that $w \models_{I'} \varphi$.
- $w \models_I \exists X \ \varphi$ iff there exists an interpretation function I' that extends I by assigning a set of events to the variable X such that $w \models_{I'} \varphi$.

A *sentence* is a formula without any *free* variables — all variables occurring in the formula are bound by a quantifier. Sentences can be assigned semantics without any interpretation function. All our policies are sentences in MSO logic.

For example, a simple access control policy over a room count context specifies that "a regular user can enter room C only if the number of regular users in C is less than the stipulated limit." We can express the formula corresponding to this policy with the following subset of Σ:
$\{C_{max}, C^d_{max}$, *request-entry-C*, *allow-entry-C* $\}$.

Formula corresponding to the above policy, which regulates access of a regular user in room C, is shown in the box in Figure 3. Semantics of the policy can be understood from the words that satisfy it. E.g. the policy specifically rules out an occurrence of C_{max} (represented by z) between the last seen C^d_{max} (represented by x) and *request-entry-C* (represented by y), if the request is to be followed by an *allow-entry-C* in a satisfying word. In English, the policy formula means that entry to the room is allowed *if and only if* there was a request for entry and the relevant context (C^d_{max} in this case) already held. Exact formulation of such rules may differ from one policy to another.

4.1.3 Template based policy description

MSO logic, being a mathematical language, makes it cumbersome for an administrator to configure complex policies with. We therefore work with a template based approach for describing policies. Details regarding facility topology and events are specified by the administrator in a simple language, which is provided as input to the offline analyzer of Figure 1. The template based configuration of policies is done such that it supports *role based access control*, wherein *roles* of users are defined based on the policies that are being enforced on them. We would like to note that *static* policies as specified using ACLs can also be specified - context becomes empty in such cases. Examples of policies for two different roles of users for our example facility are given in Table 2. Table 1 is an example of facility topology as used in Figure 1.

Different processing stages within the offline analyzer are shown in Figure 5. The *high level policy parser* entity is responsible for parsing this high-level description. As we introduced previously, contexts may comprise of two kinds of events, those events that are specific to an individual user by virtue of his/her own actions, and those events that are observed and constructed by the system for use in one or more user classes. Individual user actions comprise the history of the user. Contexts such as anti-passback and asset issuance are examples of individual user actions. The high level policy parser parses the specifications of such event declarations and usages, and constructs (predefined) regular expressions with respect to the kind of history in question. In Table 2, the policy of regular user for entry to room B requires history event $h2$. `ISSUE ASSET X IN D` implies that user must enter room D, and issue an asset `X`. Therefore, in order to enter room B, the user history must contain the sequence *request-entry-D*, *allow-entry-D*, *request-asset-X*, *issue-asset-X*. $h2$ is therefore written as an MSO logic formula representing the regular expression for this sequence. $h2^d$ implies the absence of $h2$ - the user should have either not issued the asset, or issued and returned it. Also, policy templates starting with `CAN_ENTER`, depending upon the kind of context that they use, are substituted by corresponding MSO formulae. For example, the policy of a `regular` user for room C is translated into the MSO formula of Figure 3.

Maintenance of system context, on the other hand, is the responsibility of the network of controllers. Let's assume that with respect to room B, the definition of B_{max} is similar to that of C_{max} for room C. For a regular user to enter room B, B_{max} must be false, i.e. B^d_{max} must be true. Final grant/deny decision for room B, however, is taken after evaluating both the system context and the history $h2$ of the user. The offline analyzer, therefore, must also generate configuration information for the context modeling machinery that dictates how contexts are handled. The `USES` keyword in Table 2 describes dependency of a context event on other events. Here we assume that application events `user-entry` (to a room) and `user-exit` (from a room) are events that are generated every time a user enters or leaves a room respectively. We elaborate on this while explaining our context modeling approach(Section 4.3).

The `SELF` keyword is a syntactic sugar. Note that *Regular-escort* context is defined once, but is used in connection with various rooms. We could have defined this context with respect to individual rooms in which we foresee its use. Alter-

```
# Define system context

EVENT $C_{max}$:  IS count event
      USES user-entry IN C
      USES user-exit FROM C
      PARAM_val GEQ 10
      PARAM_user-class EQ regular
      PARAM_room EQ C

EVENT $escort\text{-}Timer$:  IS timer event
      USES user-entry IN SELF
      USES user-exit FROM SELF
      PARAM_val EQ 10
      PARAM_user-class EQ regular

EVENT $Regular\text{-}escort$:  IS timed event
      USES $escort\text{-}Timer$
      PARAM_escort-class EQ regular
      PARAM_room EQ SELF
```

```
# User history based context

HISTORY $h1$:  ANTI-PASSBACK IN D
HISTORY $h2$:  ISSUE ASSET X IN D
```

```
# Role based policy for Regular users

policyclass regular:
      CAN_ENTER W on $h2^d$
      CAN_ENTER A
      CAN_ENTER B ON_CONTEXT $B^d_{max}$ AND $h2$
      CAN_ENTER C ON_CONTEXT $C^d_{max}$
      CAN_ENTER D ON_CONTEXT $h1^d$

# Role based policy for Visitors

policyclass visitor:
      CAN_ENTER W
      CAN_ENTER A ON_CONTEXT $Regular\text{-}escort$
      CAN_ENTER B ON_CONTEXT $Regular\text{-}escort$
      CAN_ENTER C ON_CONTEXT $Regular\text{-}escort$
      # Default deny for room D
```

Table 2: Template based context and policy specifications

natively, we use the keyword `SELF` for the `room` parameter, and instantiate this context for rooms `A`, `B`, and `C` at compile-time as and when it is encountered in any user's policy.

We observe that although the syntax used in our template based approach is well structured and necessary, an administrator in charge of managing the facility may find it cumbersome. For this reason a graphical user interface will be useful. This will help us to provide a practical tool to implement sophisticated policies without compromising or misinterpreting the intent. Such an interface, however, is under development at present.

4.2 Execution model

In order for the policies specified in MSO to be operational in terms of enforcing access control rules, they have to be converted into computational/executable models. These models can then be stored at appropriate locations for execution. We work with conventional finite state automata [17] as machine models for executing policies. Formal theoretical techniques are available for converting MSO formulae into automata [26]. Policies are analyzed and converted into their equivalent *finite state automata* by an *offline analyzer* algorithm (Figure 5). These automata act as rule engines executing the policies. They are constructed to allow precisely those behaviors that *satisfy* the policies. All the policies corresponding to the role of a user are collected together and converted into executable automata which are then stored in the user's smart card. Upon access request, the controller initiates execution of policies in the smart card that results in a unique access decision (allow/deny) being taken based on the response from the card.

A finite state automaton over an alphabet Σ is defined to be a tuple, $A = (Q, \Sigma, \rightarrow, I, F)$ where

- Q is a finite set of *states*.
- $I, F \subseteq Q$ are sets of *initial* and *final* states respectively, and
- $\rightarrow \subseteq (Q \times \Sigma \times Q)$ is the *transition relation*.

Semantics of finite state automata is presented in terms of its runs on a given input. *Input* is a word over Σ. Given a word $w = a_1 a_2 \dots a_n$ as input, a *run* of A on w is a sequence of states $q_0, q_1, \dots q_n$ such that $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \rightarrow$ for i varying from 0 to n. A run is said to be *valid/accepting* if $q_n \in F$. The *language* accepted by A is denoted by $L(A)$ and is defined as the set of all those words on which A has a valid run. Languages accepted by finite state automata are popularly called *regular languages* [17].

In the context of access control, a word on which the automaton has an accepting run represents a behavior of the system. Language accepted by this automaton will be the set of all possible behaviors (in terms of sequences of events) as dictated by the policies. Figure 4 shows the automaton of the MSO formula shown in Figure 3 of the `regular` user policy for room C.

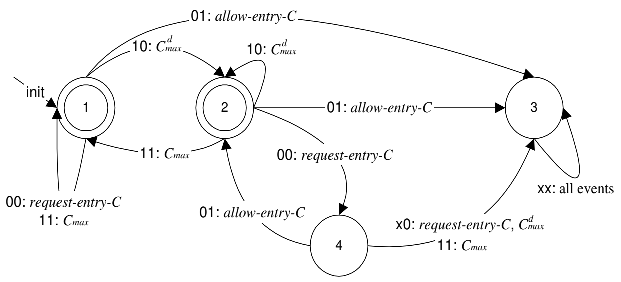

Figure 4: Automaton for a regular user policy that only depends on system context, and no individual user history

In this figure, states with double circles represent final states. Other states are rejecting states, which is to say that if a run ends on these states then the word is not accepted. The user automata are run only during the interaction of a smart card and a controller. Once a user removes his/her card from the reader, the automaton should rest in a final state. For example, starting from a final state 2, a *request-entry-C* will take the above automaton to state 4, wherefrom the controller will check for the presence of a transition to a final state upon *allow-entry-C*. In this case, it will find it and hence, open the door, and bring the automaton back to final state 2. However, while in state 2, if C_{max} event is seen the automaton will be taken to final state 1. This will happen if the occupancy of the room reaches its limit before the user can request entry. A subsequent *request-entry-C* will still keep in state 1, and now the controller will not find any final state that it can transition to on an *allow-entry-C* (note that 3 is not a final state). Hence that transition will not happen and in effect, entry will be denied, and the automaton will continue to stay in state 1.

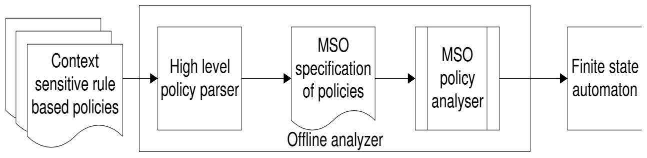

Figure 5: Part of the offline analyzer that translates input policies to execution model

Figure 5 depicts the sequence of operations that take an administrators input to an execution model. The outer box represents the offline analyzer of Figure 1. The administrator inputs (as shown in Table 2) are first translated from their high-level English-like language to an MSO specification by the high level policy parser. The offline analyzer also checks for conflicts in user policies and incorporates optimizations specific to physical access control. The MSO policy analyzer finally converts the MSO formulae into the final execution model following well-known theoretical techniques for converting formula into automata [26]. The automata finally generated, along with other initialization and configuration information, are stored on user carried devices like smart cards, as shown in Figure 1.

4.3 Context modeling

As mentioned in Section 2, the offline analyzer generates configuration information for the controllers that enables them to maintain system-wide context information. Maintaining system-wide context involves *publishing* or *subscribing* events to/from various controllers and *deriving* context events at individual controllers.

Given an enclosure like a room or a region that can have multiple controllers, we elect a single controller as the *owner* that becomes responsible for detecting and deriving context events pertaining to this enclosure. The owner of an enclosure uses local user actions and inputs from other controllers associated with the enclosure to construct most of these events. Other controllers may subscribe to these events, if such is required by the policies.

Such context events are derived using many-sorted first order predicate calculus. Ranganathan and Campbell provide one such infrastructure for context-awareness [21]. Our approach is more tightly coupled with the physical access control application. We use Prolog [11] based logic programming techniques to derive context events. All events are represented as predicates in Prolog. User and application actions are stored as facts in a knowledge base and inference rules are written to derive the predicates representing context events. The process of assertion and deletion of facts, and subsequent derivation of dependent contexts using Prolog, is a continuous one. Predicate corresponding to a context event is evaluated every time an event that it USES is asserted, retracted or derived. For example, for the context event C_{max} of Section 4.1.3, facts corresponding to user-entry and user-exit are put into the knowledge base, the predicate corresponding to C_{max} is derived from these facts whenever users enter or exit the room C.

Certain contexts events are also results of conjunctions and disjunctions of events, computed at various controllers across different enclosures as well as other devices including servers, PCs etc in the system generating individual logical events. Since every event has only one owner, any other controller can combine events through Boolean connectives without ascribing any additional semantics to the combinations, other than the combinations themselves. Note that even a Boolean combination of events will have a single owner responsible for publishing either a true or a false value of the combination at any point in time. E.g. a policy P may require events $e1$ and $e2$, detected and owned by controllers C1 and C2 respectively, in order to govern access to a particular enclosure owned by controller C3. C3 will thus need to subscribe to $e1$ and $e2$, from C1 and C2 respectively. In such cases, the administrator will define a new event $e3$ as a Boolean combination of $e1$ and $e2$, and will designate C3 to be the owner of $e3$. Deriving our requirements from the application, we note that a Boolean combination of such constituent events suffices in most cases to capture the requirements of such policies P.

The offline analyzer finds the event dependencies from the policy definitions. The controllers exchange events based on this information, through a publish/subscribe mechanism. E.g. USES keyword of Table 2 illustrates the need for subscriptions. Owners of the events are responsible for publishing the events to subscribers.

Controllers that participate in constructions of various contexts need to collaborate and hence stay connected all the time. This creates logical groups of collaborating controllers. These groups can tolerate mutual disconnect of communication without affecting the application behavior. By extension, controllers that guard rooms whose policies depend solely on user history do not require any collaboration for maintaining system context, and can thus perform in isolation. For example, for policies specified in Table 2, controllers of Figure 2 can be divided into four sets, such that controllers belonging to the same set are connected with each other and disconnected from those in other sets, viz. {C1}, {C2, C5}, {C3, C6}, and {C4}.

5. PROTOTYPING APPROACH

For the purpose of rapidly experimenting with our framework for decentralized access control, we used well-known open-source software packages, notably, MONA developed by BRICS Research Center in Denmark [16, 13] and the Java Card Development Kit provided by Sun Microsystems, Inc. [25]. These tools fit into various parts of the framework and help one to rapidly simulate, analyze and evaluate the behavior of the framework. We store events in a knowledge base and thereby reason about the presence of contexts. In this section we provide details of our prototyping approach for simple physical access control applications. We first discuss the policy analyzer, which takes high level policy and topology descriptions and converts them into a representation of the execution model. The motivation behind choosing the execution model, its representation and storage is discussed next.

5.1 Policy analyzer

We use MONA as the MSO policy analyzer of Figure 5. MONA is a tool used to translate Monadic Second-order (MSO) Logic formulae into minimal Deterministic Finite State Automata (DFA). MONA works with MSO over binary encodings. Several steps are involved in converting a given MSO formula into minimal DFA. Firstly, MONA rewrites the formula to obtain a compact representation of the same. This is followed by an inductive translation of the rewritten formula into an automaton. Finally, the automaton obtained from the above step is minimized so that an effective decision procedure can be obtained.

The high level policy parser (Figure 5) is written using a general-purpose parser generator [12] that takes administrator inputs written in a simple grammar (e.g. as shown in Table 2) and converts them to a MONA-specific MSO input form. It also translates events into appropriate binary encodings as required by MONA. During the translation process the events corresponding to requests, grants and system context, as specified by the administrator in the access control policies, are encoded as bit sequences $\{0,1\}^k$ where k is smallest integer such that for the set of events Σ defined in policy specification, $|\Sigma| \leq 2^k$. The binary encoding of the events and a basic set of predicates representing the events are first specified as templates. These are then used to construct policies that are defined by the administrator, by straightforward parsing of the input policies.

5.2 Representation of execution model

The user-carried devices need to store the automata and implement an execution program to run them. One can choose to represent the DFA in a variety of ways including adjacency matrix, linked lists, Boolean representations or even program codes with switches or if-then-else blocks. For our framework, we require a data structure that can efficiently store information pertaining to the run-time state of the application, and should be amenable to storage on small devices and fast execution. Binary Decision Diagram (BDD) [7], a data structure used to represent an automaton as a Boolean function, satisfies these requirements. A BDD is considered to be an efficient data structure for storing automata and for capturing the runtime execution mechanism of its behavior through the evaluation of next enabled transitions. The set of access control policies of a certain role, which are translated into an automaton by the policy analyzer, is stored as a BDD on user carried devices. MONA has built-in support for converting its output DFAs to BDDs.

The user carried devices also need to have an *execution program* which will efficiently traverse the BDD and compute the next state of the automaton, given an event of the system as input. One can also design the controllers to host this program since the automata are run only during the interaction of the user-carried devices and the controllers. However, we choose to store both automata (as BDDs) and the execution program on the user-carried devices for the obvious advantage of not having to communicate the BDDs themselves from the devices to the controllers. Thus, describing the application behavior in the form of policies and representing it in the form of a BDD provides a uniform framework for writing access control rules for various user carried devices e.g. smart cards, mobile phones etc.

For the sake of completion we give an example to illustrate how the state transition of an automaton translates to a traversal of the corresponding BDD representation. The execution program implements this traversal. Bryant [7] provides a detailed explanation of the working of BDDs. The finite state automaton of Figure 4, corresponding to one of the simple access control policies of the `regular` user class in Table 2, is represented as a BDD in Figure 6. As shown in Figure 4, all events (i.e. input symbols for the automaton) are represented as binary strings. Since we have only four events, we use the following 2-bit encodings, viz. *request-entry-C*: 00, *allow-entry-C*: 01, C^d_{max}: 10, and C_{max}: 11.

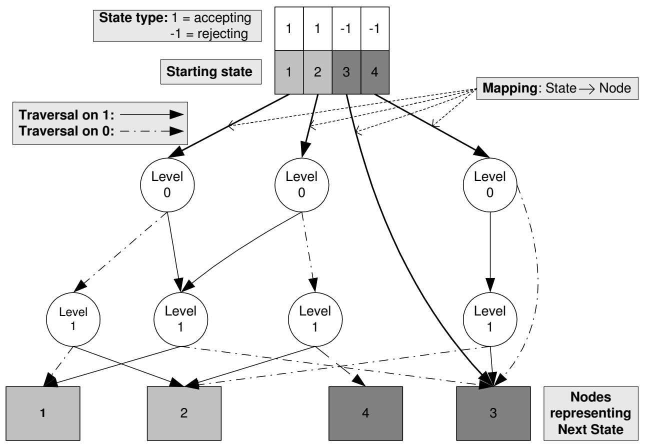

Figure 6: Binary decision diagram for simple policy

As mentioned in Section 4.2, once a user removes his/her card from the reader, the automaton must not be in a rejecting state. The automaton execution program maintains the above automaton either in state 1 or 2 between subsequent access request steps. Following the example of Section 4.2, lets assume that after a C^d_{max} the automaton rests at state 2. It is then presented with the event *request-entry-C*. Now state 2 will be the starting state for the new transition, which maps to the center node at level 0. In general, a node at i^{th} level compares the i^{th} most significant bit of the input symbol. A *request-entry-C* is encoded as 00, so the traversal takes right sub-tree (hyphenated arrows) upon comparing bits at both levels, and reaches the leaf node (box) labeled 4. This completes one transition, whereby the automaton arrives in state 4 from state 2. To complete the authorization process, the automaton execution program will now apply *allow-entry-C* and check whether or not the automaton reaches a final state. The event *allow-entry-C* is encoded as 01. Hence, starting from top right of the BDD structure at level 0 (this time from the node mapped to by starting state 4), a bit comparison takes the traversal to level 1, following the solid arrow, since the least significant bit of the encoding is 1. At level 1, upon comparison with 0, the traversal takes the left path to reach state 2 following the hyphenated arrow, since the next significant bit is 0. This is an accepting state as indicated in the top row of this BDD. Hence the operation is accepted by the execution program and the request will be followed by a grant.

Note that the generated automaton, as in Figure 4, can group certain transitions together. E.g. *request-entry-C* and C^d_{max} are grouped as x0 (x := 0,1). This means that a transition labeled by this encoding only depends upon the least significant bit. Subsequently a BDD making a level 0 comparison at starting state 4 can bypass further comparisons, if the least significant bit is found to be 0, and can directly transition to state 3. This increases the efficiency of execution. It is also evident from here that intelligent assignment of bit orderings to variables denoting various events can result in a number of grouped transitions and hence in efficient storage and execution.

5.3 Smart cards as user carried devices

In physical access control scenarios, the cost of the per-user device is likely to be an important factor for wide acceptance of this framework, given the extremely cheap passive RF-ID tags that are mostly used today. Smart cards therefore appeal as the practical device of choice to host the user-specific policies. In access control, interactions between the users and access points are typically driven by their proximity. Hence, we use Java cards [9] as the machine framework representing proximity based light weight devices.

In order to test the feasibility of our approach, we did a software prototyping of a simple set of access control policies. We used Java card Development Kit [25] to simulate Java cards, and to host BDD structure and its execution program on it. Card reader devices were simulated using the Java Card Development Kit and Open Card Framework (OCF) API [19, 20].

Together Java card and OCF provide a way of simulating the card-to-reader interactions and obtaining preliminary insights into resource requirements/limitations for hosting our access control framework based on smart cards, e.g. numbers on EEPROM usage and availability, as given by Java card development kit. Our conclusion is that with the trend of increasing memory capacities of the smart cards, meaningful physical access control solutions look viable on them. Almost all medium-end cards support 64KB currently, with promise of EEPROMS in the order of MB in the near future. The ISO 7816 standards as followed by smart cards [14], is also supported by the GSM SIM cards which leave open the possibility of using more powerful devices like mobiles phones as the proximity devices in future.

Another major advantage of using libraries and toolkits like Java card kit, which abide to standard specifications, is the assurance that they will evolve by adapting latest standards, protocols, and technologies. Such tools provide a guarantee that applications developed using them will be easy to migrate on real devices that support these standards.

5.4 Application specific optimizations

We have incorporated a few application specific optimizations to ensure that automaton state spaces are as small as possible. This allows us to target extremely light weight devices for realizing our implementation. For example, failure to find an accepting state on an "allow" transition in the automaton, implicitly means a deny in our application. Thus Σ may contain only *allow-into-A* and not *deny-into-A*, for all rooms A.

According to our execution mechanism, during the process of authorization, a controller first inputs all relevant events to the smart card, which are then applied internally to its automaton, thereby honoring the sequences of events specified by the policy. By introducing Boolean combination of events as a new event, we reduce the number of events to be communicated to the smart card and hence the decision latency. This also reduces the number of literals of the automaton, thereby reducing its size.

We note that a single complete automaton capturing all valid sequences of user movements quickly grows to sizes that are unattractive for smart cards as the number of rooms increases. In the context of this application, we instead work with an automaton per room or enclosure, without sacrificing on any meaningful transition that would have been present in a single big automaton encompassing all rooms. We did an example computation on the size of the generated automaton with the entry condition of every room comprising of a single context event and a history based condition, viz. anti-passback. While for a unified automaton approach the size increases rapidly with number of rooms, growing to 4.3MB for a modest 6 room facility, the combined size of all the automata on per-room basis increased linearly with the number of rooms reaching only 6KB for 6 rooms.

Lastly, we note that Σ of the automaton of a user doesn't usually include individual actions of other users. Although this can be incorporated as an exception, its absence helps to contain the number of automata states. In general, we capture significant user behaviors at role-level and abstract them as context events.

6. BENEFITS

A centralized architecture for context dependent authorization in physical access control does not scale with large number of users. Local decision making capacity at points of access effectively decentralizes the process of dynamic authorization. This not only reduces the per-user per-request round-trip communication to a central server, but also cuts down on the processing involved to interpret a context with respect to a user's policy owing to execution of policies on user carried devices. Further, effective decentralization and localization of policy decision enables meaningful enforcement of some access control policies even if there is a partial disconnection in the network of the controllers. E.g. policies depending only on a users past behavior, and not on other system context, can be enforced even if a controller is totally disconnected from the system. This is possible because, as we saw from the asset center example, all the relevant context information (user history) is present on the card itself. Also, re-programming of controllers for policy changes can be avoided in cases where enclosures are guarded only with the help of user-history dependent policies.

By doing away with the need to contact a central server for every access request, this framework allows and aligns itself with the growing trend of using general purpose data network in buildings and facilities. The perceived lack of reliability of such networks can assume serious proportions for centralized solutions. In our framework, the spatial locality of contexts and a local decision-making ability gives rise to only a local dependence on connectivity among the controllers. This is evident from the rooms and lobby scenario cited in Section 1, and can also be observed with respect to various other examples listed in Section 4.

Many recent access control solutions implement partial decentralization by caching ACLs locally. In this manner, access decisions for a defined region are catered by a cache, which acts as a mirror of the central repository. In cases where the number of users is extremely large, these solutions will have to keep dividing their ACLs into further localized caches. Our approach brings intelligent user carried devices within the fold of decentralization and obviates the need for data repositories for per-user context, e.g. user history based policies.

Besides, ACLs are inherently powerless to express complex policies. With the help of logic formulae we can express user behaviors in the form of regular expressions, which would comprise of a possibly unbounded number of user actions and cannot be accommodated in ACLs.

Further, in our framework, a PC or other entities can also generate and provide contexts. The framework therefore lends itself to act as a bridge between logical and physical access control domains. Controllers can stand for logical entities and generate pertinent contexts. Context thus generated is handled in the same way as other physical application contexts.

Our mechanism attempts to enforce a valid behavior by ensuring that there exists a certain sequence in the events, as permitted by the policies. The offline analyzer checks for inconsistencies and anomalies like mutually conflicting set of rules - such that only a consistent set of policies are executed in the cards. Time complexity involved in checking such sequences using automaton on the smart cards is constant, unlike most proof theoretic approaches. This is owing to the fact that our interpretation is fixed over a set of events, whereas other proof theoretic approaches use first order logic without a fixed interpretation. This coupled with the fact that in the context of the application, it is possible to optimize the size of the automaton, makes the framework appealing for smart devices.

We have chosen smart cards as representatives for these user carried smart devices in our prototyping approach — smart cards follow the widely popular ISO/IEC 7816 standard [14] that also covers the almost ubiquitous GSM SIM cards. Conformance to this standard coupled with their low cost and already existing market penetration makes them readily available for deployment. Also, improvements in hardware will ensure that memory-rich smart cards and low-cost power-efficient controllers will continue to become even more attractive for our framework of access control.

7. CONCLUSION AND FUTURE WORK

In this work we have focused on a decentralized policy evaluation framework for dynamic authorization. As an initial approach, we have simplified our assumptions about the relationship between users and roles (no role hierarchies) and derived the requirements from a specific application, namely physical access control. We observe that our specification model is useful for capturing fairly involved context-sensitive physical access control scenarios, including connecting physical access to enterprise logical access control solutions. The emphasis is on decentralized implementation, the fundamental advantage being reduction of traffic that would be otherwise required for dynamic context evaluation at a central server.

The mechanism of capturing individual role-based policies in the form of automata and storing them on user carried devices lends itself to a decentralized enforcement of these policies. Our software implementation with Java card kit establishes the viability of such a mechanism. Issues relating to performance still need to be addressed. For example, a performance analysis of communication between smart cards and card readers needs to be done. We also intend to address issues pertaining to latency of access decisions with a prototype implementation using physical devices. A few aspects concerning security are also not well addressed in our framework.

Validation of user credentials [18] and security of data transfer among various system entities are important considerations for commercial access control solutions today. While the primary contribution of this paper is to provide a solution for context based decentralized authorization, we will seek to integrate DAC with a sophisticated validation framework in future.

Decentralized context maintenance and designation of specific controllers as owners of room specific contexts introduce concerns about reliability. Ways to handle compromise of nodes and/or node failures are not explored here. With respect to the cards, tamper proofing is one way to ensure that stored data are safe from manipulation. Using cryptographic techniques to establish trust on smart cards poses another significant challenge.

8. REFERENCES

[1] M. Abadi. Logic in access control. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 228, Washington, DC, USA, 2003. IEEE Computer Society.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] M. Balasubramanian, N. Chaturvedi, A. D. Chowdhury, and A. Ganesh. A framework for rapid-prototyping of context based ubiquitous computing applications. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06)*, pages 306–311, Washington, DC, USA, 2006. IEEE Computer Society.

[4] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In V. Paxon and M. Waidner, editors, *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society.

[5] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[6] E. Bertino, E. Ferrari, and V. Atluri. A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 1–12, New York, NY, USA, 1997. ACM Press.

[7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] G. Chen and D. Kotz. Solar: Towards a Flexible and Scalable Data-Fusion Infrastructure for Ubiquitous Computing. In *UbiTools'01 - Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UbiComp'01)*, Sep. 30 2001.

[9] Z. Chen. *Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, June 2000.

[10] A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.

[11] D. Diaz. *GNU Prolog: A Native Prolog Compiler with Constraint Solving over Finite Domains*, 1.7 edition, September 2002. For GNU Prolog version 1.2.16.

[12] C. Donelly and R. Stallman. *Bison: The YACC-Compatible Parser Generator (Reference Manual)*. Free Software Foundation, Version 1.25 edition, November 1995. On-Line Info File.

[13] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[14] International Organization for Standardization. *ISO/IEC 7816*. http://www.iso.org.

[15] 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.

[16] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, Aarhus C, Denmark, January 2001.

[17] D. Kozen. *Automata and Computability*. Springer Verlag, 1997.

[18] S. Micali. NOVOMODO: Scalable certificate validation and simplified PKI management. In *1st Annual PKI Research Workshop - Proceeding*, April 2002.

[19] OpenCard Consortium. *OpenCard Framework - General Information Web Document*, second edition, October 1998. http://www.opencard.org/docs/gim/ocfgim.pdf.

[20] OpenCard Consortium. *OpenCard Framework 1.2 - Programmer's Guide*, fourth edition, December 1999. http://www.opencard.org/docs/pguide/PGuide.pdf.

[21] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput.*, 7(6):353–364, 2003.

[22] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):65–67, October 2002.

[23] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.

[24] A. Seleznyov, M. Ahmed, and S. Hailes. ADAM: An Agent-based Middleware Architecture for Distributed Access Control. In *Proc. Artificial Intelligence and Applications*, 2004.

[25] Sun Microsystems. *Development Kit User's Guide for the Binary Release with Cryptography Extensions*, October 2003. Java Card^{TM} Platform Version 2.2.1.

[26] W. Thomas. *Handbook of formal languages*, volume 3, chapter Languages, automata, and logic, pages 389–455. Springer-Verlag, New York, NY, USA, 1997.