A Model-Based Approach to Integrating Security Policies for Embedded Devices

Michael McDougall
mmcdouga@cis.upenn.edu

Rajeev Alur
alur@cis.upenn.edu

Carl A. Gunter
gunter@cis.upenn.edu

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, 19147, USA

ABSTRACT

Embedded devices like smartcards can now run multiple interacting applications. A particular challenge in this domain is to dynamically integrate diverse security policies. In this paper we show how a framework based on a concise formal model lets us securely customize a payment card equipped with a programmable chip. We present *policy automata*, a formal model of computations that grant or deny access to a resource. This model combines defeasible logic with state machines, representing complex policies as combinations of simpler modular policies. We use the model in a framework for specifying, merging and analyzing modular policies. This framework is implemented as Polaris, a tool which analyzes policy automata to reveal potential conflicts or redundancies, and compiles automata into Java Card applets.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems: *real-time and embedded systems, smartcards*; D.2.4 [Software Engineering]: Software/Program Verification: *formal methods; model checking*; D.2.11 [Software Engineering]: Software Architectures: *domain-specific architectures; languages (e.g., description, interconnection, definition)*; F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages

General Terms: Design, Theory, Verification

Keywords: Policy Integration, Model Based Design, Smartcards, Java Cards

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION

Embedded computer systems are now routinely deployed in a wide range of engineered products such as appliances, medical devices, communication devices, and automobiles. Increasingly, embedded devices, such as smartcards and cell phones, are *programmable*, and offer an open application programming interface (API) for software applications. While this offers the user the much coveted flexibility to customize and enhance functionality, it underscores the need for formal assurances about system operation as many embedded devices are used in safety-critical and security-critical contexts. We believe that the model-based design paradigm, with its promise for greater design automation and formal guarantees of reliability, is particularly relevant for this purpose. In this paper, we describe a model-based approach to adding policies to payment cards.

Smartcards are plastic cards, usually no larger than a credit card, that contain a tamper-resistant embedded processor. They are commonly used for identification, payment, and access control. Java Cards are programmable smartcards with an API that supports a restricted subset of Java (see `java.sun.com/products/javacard`). The GlobalPlatform architecture provides an extension framework for these cards, allowing installation of certified applets that run in restricted contexts or security domains (see `www.globalplatform.org`). This enabling technology, together with the obvious need for assurances of security and integrity for downloading applications on such cards, prompted us to explore formal and model-based development.

We focus on a specific form of programs called *policies*. A policy specifies whether or not a transaction should be approved, possibly based on the history of transactions. Sample policies are "the total amount of money spent during the past month should not exceed a specified limit," and "transactions involving a specified list of emergency services are always allowed." These policies can be written by multiple parties, and installed at any time. While this offers flexibility, it is necessary to detect and resolve conflicts among different policies. Also, a new policy needs to be integrated with existing policies, possibly with checks for redundancy since on-card memory is limited.

We therefore need to solve the following problem: how can we create and integrate modular security policies securely and reliably in such a way that the policies can function in an embedded environment?

The Java Card platform gives us the basic ability to combine policies which are implemented as applets written in Java. We could simply write our policies in Java and use existing Java-specific tools (for example, Java editors, type-checkers and model-checkers) to assure ourselves that our policies will behave as intended. This is unsatisfactory for the following reasons:

- A policy developer should concentrate on the core functionality of a policy—guarding access to a resource—instead of worrying about the byte-level manipulations and system calls required by the Java Card. Developers should work with a more abstract representation of policies.
- General purpose Java tools cannot exploit domain-specific knowledge to make validating a policy more efficient. Nor are general purpose tools aware of the specific problems that a policy developer is concerned with.

Our solution is a model-based approach in which we use a new formal model, *policy automata*, to define and reason about our security policies. This formal model concisely expresses the behavior with which we are concerned, while leaving other functionality to be supplied by an automatic code generator. This focus on access control and policy integration allows the developer to concentrate on correctly implementing the core functionality of an applet. Similarly, analysis tools can be optimized to check domain-specific properties. A clear formal semantics makes it easier to reason about the behavior of a policy. This approach therefore retains the ability to integrate modular policies, but it allows us to do so securely and reliably. Finally, the model is designed so that policies can easily be translated from a formal notation to Java Card applets, so the model is suitable for embedded devices.

A policy automaton is an extended finite-state machine that examines the requested transaction and votes on whether it should be accepted. Votes are written as rules in defeasible logic that essentially say which outcome the policy automaton prefers and how strong that preference is. The domain of votes is also equipped with a decision rule that combines the votes of all the policy automata and determines whether to approve the transaction, reject it, or declare a conflict. The individual policy automata update their states based on this global resolution. Using this framework one can specify policies in a modular fashion. Note that the constraints imposed by these policies are *non-monotonic* (as policies are added approval of a transaction can switch from `yes` to `no` and back to `yes`), and *stateful* (approval of a transaction depends on decisions on previous transactions). We show that static techniques such as model checking can be used to detect potential conflicts among a set of policy automata, and also to check whether a new policy is redundant with respect to a set of existing policy automata. Our policy description framework is relevant in other contexts such as firewall policies, where multiple parties wish to independently add rules governing approval of access requests.

After presenting our policy description language, we describe a prototype implementation of our approach in the tool *Polaris*. *Polaris* provides a graphical editor for specifying policies as extended state machines, and an enumerative reachability checker to detect conflicts and redundancy. We have modified the development kit from Oberthur Card Systems that allows us to install applets onto Java cards. To install a policy onto the card, *Polaris* compiles a policy automaton into a Java package, installs it on the card, and registers the new policy with the manager routine that polls all the registered policies before deciding on a transaction. We believe that this architecture for dynamically adding policies to a Java card is an advance in the state-of-the-art for smartcard technology.

The paper is organized as follows. Section 2 discusses the conflicts that arise when policies are merged and how this behavior can be modeled. Section 3 introduces our target application, programmable payment cards, and discusses the technology that makes such cards possible. Section 4 presents our formal model. Section 5 discusses our prototype tool for working with policy automata. Section 6 summarizes our contribution and discusses related and future work.

2. POLICY MERGING AND CONFLICTS

A common task for computer systems is to guard access to a resource. The policy that is used to grant or deny access is often based on a diverse set of criteria, possibly representing the interests of many different stakeholders. Describing such a policy as a combination of sub-policies may aid a developer by allowing her to focus on one piece of a policy at a time. However, when the individual policies are combined there is potential for conflicts or other interactions that make the combined policy inappropriate for its intended purpose.

Consider three policies regarding the use of a swimming pool. Each policy represents the interests of a separate stakeholder: P_{lg} is the policy put in place by the lifeguard, P_b is the policy put in place by the business administrators of the pool, and P_c is the policy put in place by the pool cleaner.

P_{lg} In an emergency no one except the lifeguard can enter the pool. The lifeguard can always enter the pool. No more than 30 people should be in the pool at one time.

P_b Nobody but the owner can enter the pool between 5pm and 9am.

P_c When 100 people have used the pool, it should be closed and cleaned.

The policies are simple to understand and are modular in the sense that each is solely concerned with the interests of the respective stakeholder. However, the policies contain potential conflicts. For example, can the lifeguard enter the pool at 6pm? A model-based approach to designing and implementing such policies will need some mechanism to reason about conflicts among stakeholders' interests.

Non-monotonic logics[5] are a family of logics in which new information may lead to previously valid conclusions being retracted.

These logics are partially motivated by a desire to capture real world common-sense reasoning. For example, if we are told that Tweety is a bird we may tentatively conclude that Tweety can fly. However, if we later learn that Tweety is a penguin we will be forced retract our conclusion. Non-monotonic logics are one possible tool for representing and analyzing the kind of conflicting swimming pool policies we see above. We can encode a rule such as "no one can enter the pool after 5pm" by marking it a tentative rule, possibly overridden if we learn more information—for example, the lifeguard needs to enter the pool because of an emergency.

The policies described above also have features that are more naturally represented as a reactive system. The decision to admit a swimmer depends on the previous events at the pool. Imagine a gatekeeper at the pool who has to decide when to let people in. If the gatekeeper cannot see the pool from where she sits she will have to keep track of how many people have entered and left the pool in order to keep the number of people in the pool below 31 (to satisfy the lifeguard) and to stop admitting people when 100 people have used the pool (so that the pool can be cleaned). So our model must have some notion of storing information and making decisions based on the history of past events.

Embedded devices like smartcards have minimal space for storing information so it is undesirable to maintain a complete history of past transactions. However, we do not want to arbitrarily restrict what information can be used to make access control decisions; we should record exactly the minimal amount of information needed by policies. In our framework we accomplish this by making the security policies responsible for collecting their own information.

In order to represent state and handle conflicts we propose a hybrid scheme for modeling interacting policies. Our model uses classical finite state automata, extended with some high-level constructs like variables, to model how policies react to and store information about previous events. We choose automata because they allow straightforward analysis and it is simple to translate them into code suitable for a smartcard. These automata interact with each other using defeasible logic [22], a non-monotonic logic designed so that statements can be proved or disproved efficiently—an important consideration if the policies must be integrated in smartcard with limited computational power. We have found that this hybrid approach succinctly models many policies that one might want to install on a programmable payment card.

3. PROGRAMMABLE PAYMENT CARDS

Payment cards such as credit cards and debit cards are a common substitute for cash and checks. There are a variety of ways in which cards come into the hands of users. A user may directly contact a bank to obtain the card, or it may be supplied to the user by an employer or parent. In the latter cases the card has a kind of 'secondary issuer' such as an enterprise or family. This secondary issuer may have policies that extend those of the bank. For instance an enterprise may stipulate that a card for an employee is used only for business expenses or a parent may stipulate that a card can only be used in an emergency. Such policies can be enforced in basically two ways in most systems. The bank or payment gateway can (and typically does) enforce certain basic restrictions such as an outstanding balance limit on the card. Other polices are enforced in a more reactive fashion by the secondary issuer when reconciling the purchase records with bills it receives. (For example, an employee can be fired or a child admonished for deviation from policy.)

A *Programmable Payment Card (PPC)* is a payment card that can be specialized with custom policies written by a secondary issuer, such as an enterprise, a family, or even the user of the card himself. PPC policies can provide privacy and risk management. For instance, in some kinds of PPC it is possible to disallow purchases before they are made on the basis of policies that are never revealed to the bank, payment gateway, or merchant. In these cases banks and payment gateways can benefit from PPCs because they shift liability for policy enforcement to the secondary issuer and user. Secondary issuers benefit by preventing some problems before admonishing or firing become necessary.

As a case study for purposes of specific analysis for policy integration we now sketch the architecture and implementation of PPCs presented in [11]. This approach is based on the GlobalPlatform implemented on Java Cards and provides for policies written in Java. These policies control payment transactions based on the Secure Electronic Transactions (SET) protocol [21]. Our PPC implementation is based on an implementation of SET by Mykhailo Lyubich [18]. There are two primary extensions. First, it is ported to run on the GlobalPlatform for the IBM JCOP Java Card simulator or the Oberthur CosmopolIC cards and, second, it is extended by a basic policy integration technique called 'simple conjunctive refinement'. In simple conjunctive refinement a collection of policies are consulted by a transaction management applet. Policies provide a boolean result and a SET transaction is allowed if, and only if, it is approved by each of the policies based on the form of the purchase request (PReq) message in the SET protocol. After it is issued, the card allows parties to add such policies but not remove them. Consequently, each new policy allows no more payment transactions than the card allowed before it was added. Policies must be approved by a certification process to ensure that they do not violate the language-based protection mechanisms of the Java Card Runtime Environment (JCRE). It is possible in principle for the JCRE to run defensively so that this step can be omitted, but this is expensive for the card. Fortunately, the policy certification only requires verifying that the program is well-formed code.

Our implementation of the simple conjunctive refinement technique was unsatisfactory for two reasons. We could not express policies which override other policies as each policy had veto power over a transaction request. Secondly, the policies were written in Java, which made it difficult to formally analyze a policy's behavior. The next section describes our formal model, which gives a more expressive policy integration mechanism and a rigorous description of policy behavior.

4. FORMAL FRAMEWORK

A *policy model* approves or rejects a transaction request based on the characteristics of the transaction request and the history of previous transactions. The model is composed of separate *policy* *automata* that vote individually as to whether a transaction request should be approved. The votes are coalesced into an approval or disapproval using a *resolution function*.

4.1 Votes and Conflicts

We use D to denote the abstract set of possible votes. Associated with D is a function f, which resolves votes into yes, no, or $\top$, representing *accept, reject,* and *conflict* (or error), respectively. As a simple example, D contains yes, no, and maybe, and f maps a set of votes to yes if the set contains yes and does not contain no; to no if contains no and does not contain yes; and to $\top$ otherwise.

For our payment card application we use defeasible logic to describe and resolve votes. We briefly introduce defeasible logic here. Readers who want a more detailed explanation of the logic are referred to [22, 19].

Atomic formulas and their negations make up the *literals* of defeasible logic. For example, $p, q, \neg p, \neg q$ are all literals. Defeasible logic has three kinds of *rules*:

Strict rules Strict rules are like normal implication:

$$\mathit{penguin} \to \neg \mathit{fly}$$

The meaning of this rule is "if *penguin* is true then *fly* is not true" (or, in other words, penguins don't fly).

Defeasible rules Defeasible rules are like strict rules except that they can be preempted by other information. For example, the rule

$$\mathit{bird} \Rightarrow \mathit{fly}$$

says that "if bird is true then we conclude that *fly* is true unless we have some reason to think otherwise."

Defeater rules Defeater rules are used to block the tentative conclusions of defeasible rules. For example, the rule

$$\mathit{injured} \leadsto \neg \mathit{fly}$$

will block a rule like $\mathit{bird} \Rightarrow \mathit{fly}$ since the knowledge that a bird is injured counteracts our intuition that birds tend to fly. However, the defeater rule (unlike a similar defeasible rule) does not lead to the conclusion $\neg \mathit{fly}$ —since we have no intuition about whether injured birds fly or not we do not want to make a tentative conclusion either way.

Each of the rules can have a set of literals on the left hand side instead of just a single literal. In such a rule all literals in the set must be true for the rule to apply.

In defeasible logic there are two notions of provability. Given a set of literals that are known to be true, called *facts*, a literal is *definitely provable* if it can be proved using strict rules and facts. A literal is *defeasibly provable* if it can be proved using facts and any of the rules. Space limitations make it impossible to include a formal description of the algorithm used to determine if a literal is defeasibly provable; readers are referred to [19].

In our framework, policies vote by giving rules that reason about a special literal yes which stands for "approve the transaction request." More precisely, there is a set of atomic formulas F which is fixed for an application. The atomic formula yes is one element of F. Let $\mathcal{R}$ be the set all rules (strict, defeasible and defeater) made of elements of F. The set D of votes is the set of finite subsets of $\mathcal{R}$. In other words, every vote $d \in D$ is a list of zero or more rules. All the votes are combined by taking the union of all the sets of rules. This combined set of rules forms the *defeasible logic theory* which we use to test the provability of the formula yes. If the votes yield a theory in which one can defeasibly prove yes without making $\neg\mathsf{yes}$ defeasibly provable then the transaction request is approved. If yes is not defeasibly provable then the transaction is rejected. If both yes and $\neg\mathsf{yes}$ are defeasibly provable (possible in defeasible logic) then there is a conflict.

4.2 Policy Models

Let T be the set of all transaction requests for a particular application domain. For example, in an e-commerce application we might have T be a set of integer-string pairs that represent the price and the seller of the transaction request. Let D be a set of votes.

A *policy automaton* P is a tuple (M, X, q_0, R, δ). The components of P are

M A finite set of *modes*

X A finite set of *variables*, each of which has a type. We write V_X for the set of possible tuples of values for all the variables in X. A state q of the policy automaton is a pair (m, v) with $m \in M$ and $v \in V_X$, and we use $Q = M \times V_X$ to denote the set of all possible states. (We separate states into variables and modes to make automaton descriptions more readable.)

q_0 An initial state (m_0, v_0) that specifies the initial mode and initial values of all the variables.

R The *rule-set* of P. R is a function

$$R : Q \times T \to D$$

which determines how the policy automaton votes in a given state to process a given transaction. Recall that D is the set of possible votes, each of which is a list of defeasible logic rules. R is called a 'rule-set' because in practice we specify R by attaching 'rules' to modes in a policy automaton.

δ The *transition function*,

$$\delta : Q \times T \times \{\mathsf{yes}, \mathsf{no}\} \to Q$$

which governs how the policy automaton updates its state when a transaction request has been approved or disapproved.

As discussed in the next section, in practice we specify a policy automaton using a graph over its modes. The edges are annotated by guards and assignments that refer to the variables and transaction parameters, and specify the transition function δ. The modes are annotated with rules that refer to the current state and the transaction parameters, and specify the function R.

A *policy model* is a triple (Π, D, f) where Π is a finite set of policy automata, D is the set of votes, and f is a *resolution function* that maps a set of elements of D to $\{\mathsf{yes}, \mathsf{no}, \top\}$.

4.3 Semantics

Consider a policy model (Π, D, f), where $\Pi = \{P_1, \ldots, P_k\}$. Let Q_i be the set of states of each policy automaton P_i. The state of the policy model at any point in time can be described by a vector $(q_1, \ldots, q_k)$, where each $q_i \in Q_i$. Initially, each policy automaton starts in its initial state. We proceed to describe how transactions are processed and states are updated.

Suppose the current state of the policy model is $(q_1, \ldots q_k)$ and the current transaction request is t. For each policy automaton P_i, its vote is $d_i = R(q_i, t)$. We then evaluate $f(\vec{d})$, where $\vec{d} = \{d_1, \ldots d_k\}$, and interpret the outcome as follows:

yes the transaction request is approved.

no the transaction request is rejected.

$\top$ there is a conflict between two or more policies.

One desirable property for a policy model is that if votes $\vec{d}$ are produced by the individual policies then $f(\vec{d}) = \mathsf{yes}$ or no—in other words, policies do not conflict with each other when composed.

Once a transaction request is approved or rejected each policy automaton updates its state. Intuitively, a policy automaton always has two possible transitions that it can follow—one to record approvals and another to record rejections. If a policy automaton is in state q and a transaction request t is approved then the state is updated to $\delta(q, t, \mathsf{yes})$. Similarly, if the transaction request t is rejected, the state will be updated to be $\delta(q, t, \mathsf{no})$.

This update extends in the natural way to states of a policy model. For a state $(q_1, \ldots q_k)$ of the policy model and a transaction t, let $d_i = R(q_i, t)$ be the vote the policy automaton P_i supplies, and let $a = f(\{d_1, \ldots, d_k\})$. If $a = \mathsf{yes}$ or $a = \mathsf{no}$, then we write

$$(q_1, \ldots q_k) \overset{t\uparrow a}{\Longrightarrow} (q_1', \ldots q_k')$$

where $q_i' = \delta(q_i, t, a)$ gives the updated state of the automaton P_i. If $a = \top$ then there is a conflict between policies and the policy model moves into a special error state $q_\top$, essentially terminating the operation of all the automata. We denote this case by

$$(q_1, \ldots, q_k) \overset{t\uparrow\top}{\Longrightarrow} q_\top$$

Once the policy model enters the error state it responds to all transaction requests with $\top$, indicating an error:

$$\forall t \in T,\ \ q_\top \overset{t\uparrow\top}{\Longrightarrow} q_\top.$$

The update relation is now generalized to a sequence of transaction requests. Given a sequence of transaction requests $\tau = t_1, \ldots, t_n$, we write

$$\vec{q} \overset{\tau\uparrow\alpha}{\Longrightarrow} \vec{q'}.$$

if there exist model states $\vec{q_1}, \ldots, \vec{q_{n-1}}$, and $\alpha = a_1 \ldots a_n$ such that

$$\vec{q} \overset{t_1\uparrow a_1}{\Longrightarrow} \vec{q_1} \overset{t_2\uparrow a_2}{\Longrightarrow} \cdots \overset{t_{n-1}\uparrow a_{n-1}}{\Longrightarrow} \vec{q}_{n-1} \overset{t_n\uparrow a_n}{\Longrightarrow} \vec{q'}.$$

Given a policy model A and a sequence τ of transaction requests we say A *emits* α *on* τ if for the initial state $\vec{q_0}$ of the model, there exists some $\vec{q'}$ such that

$$\vec{q_0} \overset{\tau\uparrow\alpha}{\Longrightarrow} \vec{q'}.$$

4.4 Conflicts

A policy model with initial state $\vec{q_0}$ is *conflict-free* if for all sequences τ of transaction requests, $\vec{q_0} \overset{\tau\uparrow\alpha}{\Longrightarrow} \vec{q'}$ implies $\vec{q'} \neq q_\top$. It is easy to see that a conflict-free model will never emit $\top$ in response to a transaction request. Typically a developer will want to ensure that her policy model is conflict-free before deploying it.

4.5 Redundancy

Intuitively, a redundant policy automaton is one which has no effect on the responses to transaction requests.

Given a policy model $A = (\Pi, D, f)$ where $\Pi = \{P_1, \ldots, P_k\}$ then policy automaton P_i is *redundant in* A if for all sequences τ of transaction requests, A emits α on τ if and only if the policy model $(\Pi - \{P_i\}, D, f)$ emits α on τ.

In some circumstances having a redundant policy automaton may be undesirable—it may be an indication that a policy is being overridden by other policies. At the very least, it indicates that a simpler, smaller model could be used to do the same job. If a device has a limited amount of memory in which to store programs then a developer would want to avoid installing redundant policy automata. However, if a policy automaton P is redundant with respect to a policy model $A = (\Pi, D, f)$ it may not remain redundant if we add some policy automaton P' to Π. A developer may therefore want to install a redundant policy automaton on a device if she expects more policy automata to be installed on the device in the future.

5. PROTOTYPE

We are implementing a prototype called *Polaris* (`www.cis.upenn.edu/~mmcdouga/polaris`) that performs policy automata analysis and compilation. It includes a graphical interface for editing the automata, an analysis engine that checks for policy conflicts, and a code-generator that creates Java Card applets that implement the policy automata. The architecture of the prototype is shown in Figure 1. The tool is being implemented in Java and uses the Hermes [1] code base. The prototype has four modules:

Front end: A developer uses the graphical front-end to create, edit and save policy automata. The automata are described using a graphical language made up of boxes and arrows which are annotated with small pieces of text; creating automata is much like using a graphics application like xfig or Adobe Illustrator. The automata are stored as XML. The front end must also interact with the analysis engine to illustrate the outcome of any analysis procedures. Figure 2 shows a screen shot of the automata editor.

Analysis engine: The analysis engine takes a policy model from the front end and checks that the automata satisfy various properties the designer chooses: conflict-freedom, reachability of certain states or whether an automaton is redundant with respect to other

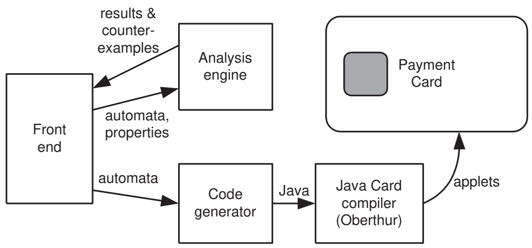

Figure 1: *Polaris* architecture

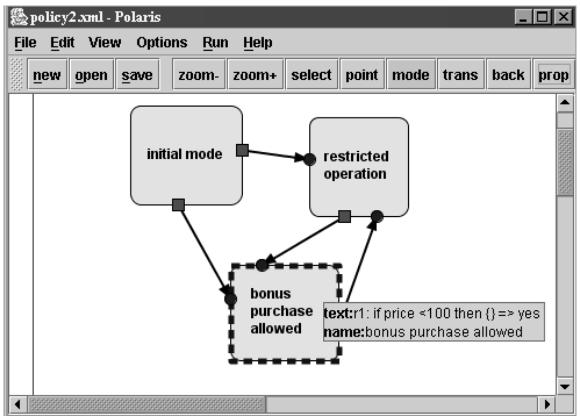

Figure 2: *Polaris* automata editor

automata. The analysis algorithms are discussed in more detail in Section 5.3. The analysis engine borrows some code from the enumerative reachability procedures of Hermes but is still only partially complete.

Code generator: The code generator converts a policy model into Java that is suitable for a Java Card. Each policy automaton is compiled into a separate applet that implements that policy. This architecture of separate applets allows new policy applets to added to the card dynamically.

Payment card: The payment card provides the run-time environment for the policy automata that have been compiled into Java Card applets. The payment card takes part in a SET transaction with a remote website via a local PC that has a Java Card reader. Before the transaction takes place the policy model implementation must approve the purchase request.

5.1 Graphical Language

Polaris uses a graphical language to describe policy automata. A policy model is created by drawing a number of rectangles, each of which represents a policy automaton. Each of the policy rectangles can be annotated with a list of variables X that store information needed by the policy automata. Inside those rectangles, the developer can draw rounded rectangles which represent the policy automaton's modes. Figure 2 shows a policy automaton with three modes being edited in *Polaris*.

The δ transition function is specified by drawing arrows from one mode to another. Each arrow is annotated with yes or no, indicating whether the transition should apply to an accepted or rejected transaction request, and a boolean expression involving the variables of the policy automaton and the transaction, and a list of updated values for the variables. The boolean expression is similar to the expressions in high-level programming languages like Java or C. For example, a transition from m to m' could be annotated with yes and the expression "`t.price<30 & count==1`", where *count* is a variable and t is a transaction request, and variable update "`count:=2`". Such a transition gives a partial description of δ, mapping $((m, v), t, \text{yes})$ to (m', v') for all variable settings v where $count = 1$, for all t with a price under 30, and where v' has the same variable settings as v except that *count* is now 2.

The rule-set function R is specified by annotating the mode rectangles with *rules*. Each rule has a boolean expression (like the expressions attached to the transition arrows) referring to the current transaction request and the variables of the automaton, and a vote d. If a policy automaton is in a mode m which is annotated with rule r and a transaction request arrives that, along with the current variable settings, makes the boolean expression true, then vote d becomes the policy automaton's vote. Votes are lists of defeasible logic rules written in the syntax of the Deimos defeasible logic query tool [20]. Each rule therefore gives a partial description of R. Figure 2 shows a list with one rule that has been attached to the "bonus purchase allowed" mode. The expression is "`price < 100`" and the vote is "`{}=> yes`", which is $\{\} \Rightarrow \text{yes}$ written using ASCII characters. The rule essentially says "conclude yes tentatively unless others override."

We use simple typing rules to check if expressions involving policy automaton variables and the transaction requests are well typed. Each variable must be declared as a particular type (for example, a boolean, integer or enumerated type). Transaction requests are treated as records with a number of fields, each of which has a particular type. We check that types are used consistently—for example, an integer is not compared to a symbol or a boolean variable is not set to 3. We also perform checks on the graphical structure to ensure that the picture on the screen can be translated into a policy automaton.

5.2 Example: a Payment Card Policy

We now show an example of a policy model made up of the following policies:

P_3 Allow up to 3 purchases per day.

P_E Guarantee payment to emergency services twice.

P_{cc} A cash card: spend no more than \$500 total.

P_N No alcohol can be purchased.

P_t Prevent purchases of prescription drugs which conflict with the anti-depressant Tofranil.

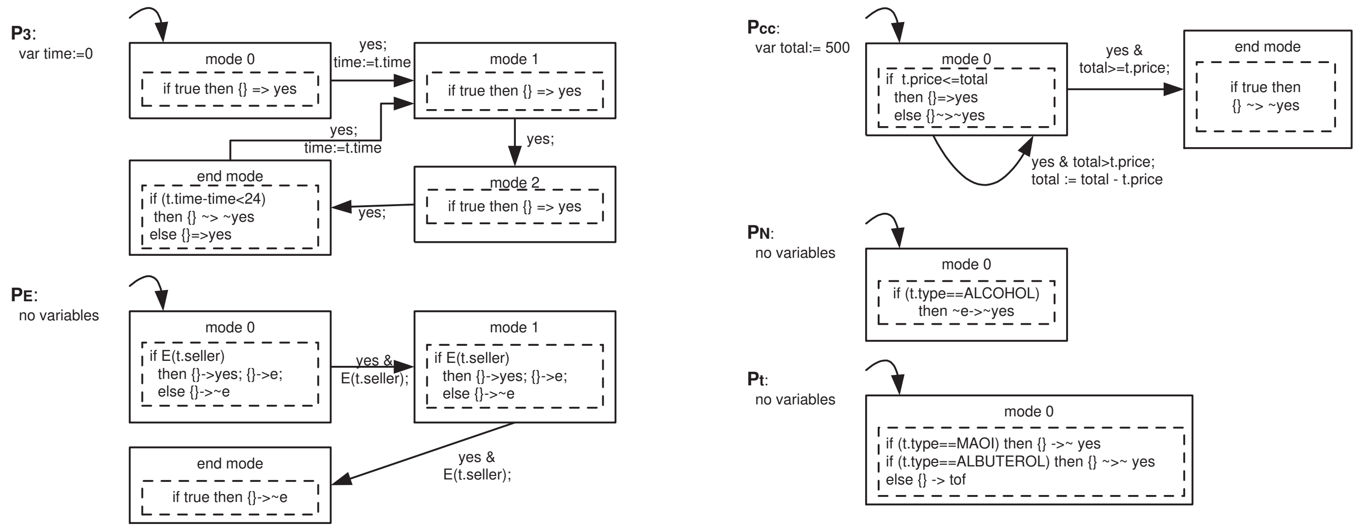

Figure 3: Example payment card policy model

The last policy, P_t, deserves some explanation. Tofranil is an prescription drug used to treat depression. It can be fatal when combined with a drug that is a monoamine oxidase inhibitor (MAOI). We envision P_t being installed by a doctor or a pharmacist when the card holder begins taking Tofranil. This policy will prevent purchases of drugs that conflict with Tofranil, thereby reducing the risk that a mistake by a doctor or pharmacist leads to a fatal drug interaction. Tofranil can also interact with another drug called Albuterol, but the interaction is less severe so our policy automaton is not as insistent about rejecting purchases of Albuterol.

Figure 3 shows these five policy automata in a simplified form of the graphical language accepted by our prototype. Variables are declared at the left of the diagram, along with the initial value of the variable. For example, the initial value of P_{cc}'s variable `total` is 500.

Modes are indicated by rectangles with solid lines. A mode's rules are contained in a rectangle with a dotted border within the mode. Rules are written in the form "`if` *expression* `then` *vote*". The expression `E(t.seller)` used in the rules of P_E is a predicate that is true if `t.seller` is contained in a set of approved emergency service sellers (for example, hospitals and ambulance companies). The word ALCOHOL in the rule of P_N refers to a standard product identifier that identifies a purchase as alcohol. Similarly, the words MAOI and ALBUTEROL in P_t refer to standard identifiers for particular classes of drugs.

The rule's *vote* is written as a list of rules of defeasible logic. We describe a few of the votes that appear in the example here.

{}=>yes the transaction request should be approved tentatively but can be overridden

{}~>~yes override a tentative approval

{}->yes;{}->e approve the transaction and assert that the literal `e` is true. Making `e` true signals to other automata that the transaction request is an emergency.

~e->~yes if `e` is not true then reject the transaction request. This vote allows P_N to override P_3 and P_{cc} without conflicting with P_E.

When no rule applies in a given state then an empty set of defeasible logic rules is used as the vote.

As described above, arrows represent transitions between modes. The annotation attached to the arrow has the form "*expression* ; *update*". The *expression* indicates when that transition is enabled and the *update* section determines how the variables are updated. For example, in P_{cc} the transition with an expression "`yes & total == t.price`" is enabled when a transaction request has been approved and the total is equal to the transaction price. If the update section is empty then no change will be made to the variables. When there is no enabled arrow starting at a mode then no update is made to variables or modes when the transaction request is approved or rejected. For example, if P_{cc} is in mode 0 and a transaction request is rejected then the variable `total` is left unchanged and the automaton stays in mode 0.

We now quickly sketch how the policies in figure 3 react when given the following sequence of transaction requests: t_1, a $40 alcohol purchase which is not an emergency; and t_2, a $300 bicycle purchase. The request t_1 has its 'type' field set to ALCOHOL so policy P_N will vote `~e->~yes`, while P_E will vote `{}->~e` because the request is not from an emergency seller (i.e. `E(t.seller)` is false). The defeasible logic engine will recognize that these two votes form a proof of `~yes`. Policies P_{cc} and P_3 both contribute `{}=>yes` as votes, but this defeasible rule is overridden by the strict rule in P_N's vote. Policy P_t contributes a vote

$\{\}$->tof, but this vote does lead to a proof of yes or $\sim$yes. Since $\sim$yes has been defeasibly proved and yes has not been proved we reject the transaction. All the arrows in our policies are enabled only when a transaction is accepted so no updates are made to variable or modes after the first transaction request is rejected.

When t_2 is submitted the policy P_{cc} supplies the vote $\{\}$=>yes because the price of $300 is below the value of the variable total, which was set to 500. P_3 submits the same vote as P_{cc}. Since this purchase does not involve alcohol the policy P_N has no specific vote—a default empty vote (i.e. a zero-length list of defeasible logic rules) is therefore submitted. P_E submits the vote $\{\}$->$\sim$e since the seller is not an emergency seller. Policy P_t again submits $\{\}$->tof since the purchase involves neither Albuterol nor an MAOI. The defeasible logic engine will show that yes is defeasibly provable since no votes overrule P_{cc}'s vote. Nor do any votes conclude $\sim$yes so the transaction is approved. This triggers P_3 to move from mode 0 to mode 1 and update its time variable to the time of the transaction. P_E will not change modes because the seller is not an emergency seller. P_{cc} will stay in mode 0 but it will change the value of its variable total from 500 to 200. P_N and P_t each have one mode and no variables so they do not update their state.

5.3 Analysis

If the types of the variables are finite then a policy model must be in one of a finite number of states. For infinite types we can make the number of states finite by using abstraction. We can therefore use a conservative on-the-fly reachability analysis to look for states where conflicts occur. If none of the reachable states will emit $\top$ on any transaction request then we know that our model is conflict-free.

Checking a given state for conflicts involves evaluating the resolution function f on all possible combinations of votes in that state. Computing f can be done efficiently as [19] gives an algorithm for finding the consequences of a defeasible theory in time that is linear with respect to the number of literals and defeasible logic rules.

We may also want to check for redundant policies. For a given model state $\vec{q}$ let $d_{\vec{q},t,i}$ be the vote that the i-th policy automaton gives when processing transaction t in state $\vec{q}$. The policy automaton P_1 is *redundant at* $\vec{q}$ if

$$\forall t \in T, f(D_{\vec{q},t}) \quad = \quad f(D'_{\vec{q},t}) \qquad (1)$$

where $D_{\vec{q},t} = \{d_{\vec{q},t,i} | \; i = 1 \ldots k\}$ and $D'_{\vec{q},t} = D_{\vec{q},t} - \{d_{\vec{q},t,1}\}$.

P_1 is redundant in $(\{P_1, \ldots, P_k\}, D, f)$ if it is redundant at each reachable model state. We can therefore check for redundancy by finding all reachable model states and verifying that each state satisfies equation (1). As discussed above, evaluating f for all transactions can be done efficiently.

5.4 Code Generation and the Java Card Platform

There are two types of Java Card applets that need to be generated: the *manager* applet and the *policy* applet. The manager applet is responsible for polling the policy applets for their votes, consolidating the votes to decide whether the transaction request should be approved, and then notifying the policy applets about the approval or disapproval. There is one manager applet on a programmable payment card and it must be installed before any of the policy applets. Our prototype applet is based on the Lyubich's SET implementation [18] and most of the applet is concerned with the details of the SET protocol. However, we have added a defeasible logic engine to the applet so that it can process the votes of the policy applets. Most of the manager applet's code deals with Java Card and SET protocols; this code is specified as a template that is constant for all manager applets. We envision different applications using different transaction request types (for example, the transaction date may be available in some applications and unavailable for others) so we automatically generate the manager applet code that processes the transaction request data.

The Java Card platform imposes certain constraints on the applet implementation. Garbage collection is not available on most cards, so care must be taken to allocate the minimal memory necessary. All data must be stored as 8 or 16 bit values. Unlike the standard Java platform available on desktops and servers, a Java Card has two kinds of memory: RAM and EEPROM. RAM is like the RAM in most computers – it can be read from and written to quickly, and it loses its data when power is cut off (for example, when a card is withdrawn from a card reading terminal). Due to cost and size constraints, RAM is limited to 1 or 2K in the currently available cards. EEPROM will retain data when power is lost, and it is cheaper than RAM so it is feasible to put as much as 64K on a single card. However, EEPROM can only be written to a limited number of times (typically on the order of 100,000) and writes are slow, so EEPROM should not be used for memory which is updated frequently.

Our on-card defeasible logic engine (DLE) needed to account for these restrictions. The DLE needs to compute all the literals that are defeasibly provable given a defeasible logic theory. We partition the memory required for the algorithm into two parts: stable and volatile. Stable data is kept in EEPROM and volatile data is kept in RAM. Our algorithm keeps the rules of the theory in stable memory, while using volatile memory to track the proof status of each of the literals in the theory. While the total memory required by the DLE is proportional to the size of the theory, the volatile memory required is proportional to the number of literals in the theory. To conserve EEPROM memory, we keep only a single copy of the rules in the defeasible logic theory. This copy is maintained by the policy applet which is supplying the vote which contains the rule.

A policy applet implements a single policy automaton. Many policy applets can be installed on the same card. Starting from a template applet, the code generator adds two methods getVote and update, which return a vote and update the state of the applet, respectively. The set of all possible votes is computed by the code generator and each vote is instantiated as a member variable stored in EEPROM. We precompute this set of votes to minimize the amount of RAM required at runtime. Examples of the output of this code generation are available at the *Polaris* web site (www.cis.upen.edu/~mmcdouga/polaris).

A smartcard's limited memory makes code size an important consideration. Table 1 shows how much the code size increased

	CAP file size	methods	static fields
original SET (bytes)	11291	3715	3355
modified SET (bytes)	15586	5911	3591
increase due to modification	38%	59%	7%

Table 1: Code size for original and modified SET manager applet

	CAP file size	methods	static fields
P_3 (bytes)	670	364	26
P_E (bytes)	707	376	26
P_{cc} (bytes)	645	342	26
P_N (bytes)	579	296	24
P_{cc} (bytes)	639	339	28

Table 2: Code size for selected policy applets

for the Java Card implementation of the SET protocol when we extended it to use our policy integration architecture. The second column of the table shows the size of the converted applet (CAP) file. CAP files are the standard package format for Java Card applets. The table also shows the number of bytes required to represent the methods (executable code) and static fields (persistent data) of the applet; these two components are the largest components of the CAP files. After extending the SET applet with a defeasible logic engine and the code necessary to manage policy applets the total applet size is only 38% larger. Table 2 shows the size of the five applets generated from the automata in Figure 3.

5.5 Adding Policies Dynamically

The policy model gives developers a formal framework for combining the policies of different stakeholders. Different departments in an enterprise can each create their own modular policies and when these policies are installed on a card they can be checked against each other to ensure that they are, for example, conflict-free. This increases the assurance that a payment card will behave properly when given to a user. However, the Java Card/GlobalPlatform architecture allows new applets to be installed *after* the card has been issued. In this section we discuss how our framework can be adapted for the case where arbitrary parties, who may not be affiliated with the enterprise that issued the card, wish to add new policies. We call the set of policies that are initially installed the *base policies*. The policies added later are called the *supplemental policies*.

In order to allow new policy automata to be checked with respect to previously-installed policies we require that an installed policy provide a way to access its policy automaton. This can be stored on the card or referenced by a URL. A developer will compose these policy automata with her new policy automaton and check that the new policy automata is conflict-free (or whatever property is desired). If the desired properties hold, the developer follows the steps described in [11], which exploit the GlobalPlatform security model. She generates valid JCVM byte code and supplies it to a certification authority, who uses it to generate a CAP file with a digital signature. The CAP file, together with signed load and install instructions, are then supplied to the developer who uses them to load and install the new applet onto the card. The digital signatures protect the card from the installation of invalid CAP files. When the new applet is selected (a basic Java Card operation that chooses a particular applet for execution), it registers itself with the manager applet installed by the primary issuer. If the applet is subsequently removed, the manager applet disables the card.

In order to protect the functionality of the base policies from policies that were not analyzed we modify the resolution function slightly. If the updated set of applets generates a $\top$ then we fall back to the base automata and evaluate f using only the votes from the base policies. Since the base policies were installed before the card was issued we can be confident that they are conflict-free. Once the transaction request is approved or rejected, *all* policy automata (base and supplemental) update their state and continue as if the conflict had not occurred.

6. DISCUSSION AND CONCLUSION

Our work makes three contributions. We describe a novel application: programmable payment cards with a dynamic on-card policy management framework. We introduce *policy automata*, a formal framework that combines state machines with defeasible logic, which models the dynamic integration of modular policies. Finally, we have implemented *Polaris*, a suite of tools that integrates design, analysis and compilation for policies expressed as policy automata.

6.1 Related Work

This work builds on a wide range of previous work in formal methods [8], especially in model-checking [7] techniques and tools such as SPIN [16]. Using state-machine-based models for high-level designs is quite common in software engineering (e.g. Statecharts [13], UML [3]). Easterbrook and Chechik [6] analyze merged state machines by using paraconsistent logics to capture the possibly inconsistent views of the system. Siddiqi and Atlee [24] use a hybrid model that combines state-transitions and logical assertions to model and analyze feature interaction conflicts in telephone systems. Lupu and Sloman [17] discuss a number of strategies for resolving policy conflicts. There is related work using non-monotonic logics for reasoning about policies. Grosof et al. [10] represent business rules using courteous logic programs, while Antoniou et al. [2] use defeasible logic to represent administrative regulations governing, for example, exam scheduling. Various policy specification languages have been proposed. Damianou et al. [9] use the Ponder language to describe access control policies. Hoagland et al. [15] use a graphical language to describe security policies. Miro [14] also uses a graphical language and allows policies to override other policies. Halpern and Weissman [12] propose using a fragment of first-order logic as a security policy model which accommodates merged policies and has a tractable algorithm to determine access rights. These approaches target a wide range of access control policies protecting many resources while ours is concentrated on protecting one resource. It is not clear whether that they are suitable in an embedded context. As far as we know, there is no prior work on combining state-machine based modeling, non-monotonic logics, and formal analysis.

In recent years there has been a lot of research on formal methods for Java Cards [4]. This research typically focuses on proving correctness of protocols and API implementation. The problem of adding policies dynamically and merging them with existing policies has not been addressed.

Schneider [23] uses *security automata* to model access control policies and generate monitors that enforce correct behavior. Schneider's security policies are primarily intended to constrain programs while our policies constrain users. We use a voting system to integrate different policies instead of simply taking a conjunction of policies.

6.2 Future Work

We plan to extend this work in a number of directions. We will continue refining and extending our tool to explore heuristics and other engineering issues involved in analyzing policy automata and generating code that implement the automata. A more rigorous evaluation of the tool will be performed in order to quantify the efficiency of various analysis strategies and the on-card running time of the applets.

We think that aspects of this work will be applicable for guarding access to network resources. In particular, we will examine whether our policy model can adequately express the policies governing network packet processing and forwarding in firewalls. Similarly, policy automata look promising as a model for representing HTTP access policies.

The formal aspects of this work can be extended in various directions as well. One possible extension would be to modify the policy model so that transactions requests would yield more than yes and no as answers. For example, a request to access a file might yield yes-read-only as an answer in addition to yes and no. Policy automata as described here get only one chance to react to a transaction request. However, there are applications where a policy automaton may want to react to the outcome of a transaction that has been approved. For example, a cell phone policy governing what phone numbers may be called may want to react one way when an outgoing call where the other party fails to pick up the phone, and another way when the other party picks up the phone and has a conversation. The set of votes D and the resolution function f are abstract parameters in the definition of a policy model. In this work we use defeasible logic for D and f but we could replace them with some other voting system based on a more expressive non-monotonic logic (such as default logic or abductive logic), deontic logic, or multi-valued logic.

7. ACKNOWLEDGMENTS

We would like to thank Watee Arjsamat, Alwyn Goodloe, Mykhailo Lyubich and Jason Simas for their work on the programmable payment card prototype and for many helpful discussions.

This research was supported in part by ARO URI award DAAD19-01-1-0473, and NSF award CCR-0209990.

8. REFERENCES

[1] R. Alur, M. McDougall, and Z. Yang. Exploiting behavioral hierarchy for efficient model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 338–342. Springer-Verlag, 2002.

[2] G. Antoniou, D. Billington, and M. J. Maher. On the analysis of regulations using defeasible rules. In *32nd Annual Hawaii International Conference on System Sciences (CD/ROM)*. Computer Society Press, 1999.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] C.-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. Technical Report NIII-R0316, University of Nijmegen, Department of Computer Science, Sept 2003.

[5] G. Brewka, J. Dix, and K. Konolige. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[6] M. Chechik, S. Easterbrook, and V. Petrovykh. Model-Checking over Multi-valued Logics. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity International Symposium of Formal Methods Europe*, pages 72–98. Springer Verlag, 2001.

[7] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[8] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In M. Sloman, editor, *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY), LNCS*, volume 1995, pages 18–38, 2001.

[10] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in xml. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 68–77. ACM Press, 1999.

[11] C. A. Gunter. Open APIs for embedded security. In L. Cardelli, editor, *Proceedings of the European Conference on Object Oriented Programming*, July 2003.

[12] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 187–201, 2003.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] A. Heydon, M. W. Maimone, J. D. Tygar, J. M. Wing, and A. M. Zaremski. Miro: Visual specification of security. *IEEE Trans. Softw. Eng.*, 16(10):1185–1197, 1990.

[15] J. A. Hoagland, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. Technical Report CSE-98-3, University of California, Davis Department of Computer Science, 1998.

[16] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[17] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, 1999.

[18] M. Lyubich. Die architekturen von SET mit der Java Card. In A. Bode and W. Karl, editors, *ITG Fachbericht, APC 2001 Arbeitsplatzcomputer*, 2001.

[19] M. J. Maher. Propositional defeasible logic has linear complexity. *Theory and Practice of Logic Programming*, 1(6):691–711, 2001.

[20] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001.

[21] Mastercard and Visa. *SET Secure Electronic Transaction Specification: Formal Protocol Definition*, May 1997.

[22] D. Nute. Defeasible logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 353–395. Oxford University Press, 1994.

[23] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[24] S. Siddiqi and J. M. Atlee. A hybrid model for specifying features and detecting interactions. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 32(4):471–485, 2000.