MODELING AND ANALYZING INTEGRATED POLICIES

Michael McDougall

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2005

Carl A. Gunter and Rajeev Alur
Supervisors of Dissertation

Benjamin C. Pierce
Graduate Group Chairperson

Acknowledgments

This dissertation would have been impossible without the help and support of many.

This work was built on a foundation that was established by the OpEm group at Penn: Watee Arjsamat, Alwyn Goodloe, and Jason Simas' early work on the programmable payment card prototype gave me a big head start. My work would have been much tougher if they had not already mapped out many of the dead-ends that loom when working with new technology. Alwyn Goodloe deserves special mention as a collaborator on many projects in addition to the payment card work—there were many occasions where my work depended on the product of his sweat and tears.

The policy automata work was the product of many discussions with my my advisors, Carl Gunter and Rajeev Alur. I would like to thank them for all their help and advice on this project as well as many other projects throughout my time at Penn. They were always ready to do what needed to be done for me. They guided me to research projects when I started. They eased the bureaucratic hassles of being a graduate student. They shared their wisdom on conducting research, from finding the right research strategies to getting the LaTeX looking right. Finally, when the time was right, they knew when to step back and let me struggle on my own. Through all this they became my friends. I am flattered and honoured by the trust they have given me.

This dissertation was greatly improved by the comments, questions and suggestions from my dissertation committee: Insup Lee, Andre Scedrov, Jeannette Wing and Steve Zdancewic. I thank the committee members for their attention and enthusiasm, especially Dr. Wing for making the effort to travel to Philadelphia to attend my defense.

Penn has a great faculty and I came to feel like I was supported by a large group of tertiary advisors. Two standout examples of this group are Benjamin Pierce and Insup Lee. Benjamin Pierce inspired me through his teaching, research and good-natured enthusiasm. He demonstrated how one can pursue rigourous theory and cool technology. Insup Lee's endless skepticism as a teacher and researcher forced me to really understand what I was talking about a number of times—a lesson I have tried to internalize—and his advice has been helpful and reassuring.

I would also like to thank Michael Felker, my department's Graduate Coordinator, who, time and time again, cut a safe path through bureaucratic jungles.

In my early years at Penn I was surrounded and inspired by the folks in the Theory Lab: Trevor Jim, Davor Obradovic, Pankaj Kakkar and Karthik Bhargavan. Those presentations and brainstorming sessions prepared me for my research. I would like to especially thank Trevor Jim for his help, sometimes offered directly and sometimes by standing as an example of someone with a great eye for good research.

My life as a graduate student was eased by selfless financial, practical and moral support from my parents and my parents-in-law.

Most of all I need to thank my wife, Megan Susnis. Her support and patience as four years stretched into six and a half, and as her husband sunk deeper into a dissertation, was enormously valuable to me. She endured disruption and long periods of uncertainty so that I could study what I love. I hope I can pay that debt back some day.

ABSTRACT

MODELING AND ANALYZING INTEGRATED POLICIES

Michael McDougall

Carl A. Gunter and Rajeev Alur

Smart card technology has advanced to the point where computerized cards the size of credit cards can hold multiple interacting programs. These multi-applet cards are beginning to be exploited by business and government in security, transport and financial applications. We conduct a thorough analysis of a programmable payment card application: a smart card for making purchases which can be customized to allow or reject purchases based on various policies that are installed by users. We describe a framework for specifying, merging and analyzing modular policies. We present *policy automata*, a formal model of computations that grant or deny access to a resource. This model combines state machines with a voting system whereby the vote of each state machine is consolidated and resolved into a decision to accept or reject. We use defeasible logic as the primary mechanism for describing and resolving votes. This formal model effectively represents complex policies as combinations of simpler modular policies. We present Polaris, a tool which analyzes policy automata to reveal potential conflicts and compiles automata into an executable form when combined with our on-card policy manager. We show the effectiveness of our model in a case-study where actual University of Pennsylvania purchasing policies are encoded as policy automata. We demonstrate the feasibility of our framework with experiments that show that our implementation can convert formal policy automata to executable Java Card applets whose performance meets the requirements for retail credit card transactions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As computer chips get smaller, cheaper and more powerful they are working their way into everyday items and appliances. This emerging world where everything is equipped with a computer has enormous potential for offering users new functionality and flexibility. However, care must be taken to ensure that this new flexibility does more good than harm; if everything we own is managed by computers then a bug or misconfiguration can have serious consequences.

The *Open Embedded Systems* (OpEm) project at the University of Pennsylvania (Penn) (`http://securitylab.cis.upenn.edu/opem/`) explores the intersection of modularity, flexibility, dependability and predictability issues that arise when we exploit the new functionality of *embedded devices*—machines, appliances and everyday items augmented with computers. If we want to truly exploit the functionality of these small computers we need to design interfaces that allow users to perform sophisticated configurations, run scripts or even programs on the devices; in other words, the devices must be modular and flexible. Since embedded systems are often used to protect or manage critical resources, these configurations, scripts and programs must be well behaved; we need techniques to ensure they are dependable and predictable.

One concrete example of this need is found in the OpEm group's project on *programmable payment cards*. Users can add their own payment restrictions to the cards to protect against accidental or malicious use by themselves, friends, children or employees. A recent Wired News article about the project included comments from a member of payment card industry:

> Tony Mitchell, vice president of public relations with American Express, said the technology could be a hit with users.
>
> “I’d imagine that some people would want that level of control and flexibility,” Mitchell said. “It would add another dimension.”
>
> *Wired News, November 7, 2003*

This flexibility brings risks with it. A user who installs a new payment policy will want some guarantees about the behavior of the modified card. Will the new policy really take effect, or will it be overridden by some other policy? Will the new policy damage the card or behave in other undesirable ways?

This dissertation addresses this need for flexibility and predictability in one family of applications: those that control access to a critical resource. In this work we focus on the programmable payment card application, though we think the techniques could be applied in other applications such as IP packet filter rules or database access rules. This work establishes a formal framework that helps us understand and reason about systems where new policy rules are added to existing policy rules.

We integrate this formal model with a prototype development environment, called Polaris, to enable *model-based design*—a design paradigm where the same formal model is used for analysis and as a basis for implementation.

1.1 Modeling 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 except 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 if there were some kind of emergency? A model-based approach to designing and implementing such policies will need some mechanism to reason about conflicts among stakeholders' interests.

Non-monotonic logics[11] 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 [56], a non-monotonic logic designed so that statements can be proved or disproved efficiently—an important consideration if the policies must be integrated in a 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.

1.2 Scope of the Work

We have the following goals:

- A succinct formal model that can describe stateful, potentially conflicting policies. The model should allow us to formally define intuitive properties of policies—for example, conflict-freedom and redundancy.

- Techniques for analyzing this formal model in order to determine whether a given instance satisfies desirable properties.

- A framework for using this model in model-based design. That is, to use this model as a formal description that is both amenable to analysis and a source language that can be used to generate an implementation.

- A working implementation of the framework.

1.3 Limitations of the Previous Research

The current state of the art concerning formal models and analysis of security policies is unsatisfactory for a number of reasons. There has been extensive research in formal models of computation and formal analysis of those models but we are not aware of any model that succinctly captures the key features of the interacting policies we would like to model.

There are formal models of network access control, especially firewall rules, but these are too restrictive to capture the stateful policies required on a programmable payment card. Traditional state machine models typically combine state machines by taking a conjunction of separate state machines and do not model policy interactions and conflicts very well. Non-monotonic logics can model policy interactions elegantly but are clumsy when modeling stateful, reactive behavior. Furthermore, most non-monotonic logics are not obviously suitable for a platform with limited computational power. We see a need for a new formalism that efficiently models the behavior and properties of stateful access control policies.

Existing analysis techniques are also insufficient 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.

Another approach would be to use an existing special-purpose language that is designed for access policies and is amenable to analysis. We are not aware of any suitable language; existing policy languages are either not amenable to analysis, or they are too application-specific to faithfully model the stateful policies we are interested in (for example, firewall analysis tools), or they do not handle modular addition or subtraction of policies, or are not suitable for a limited platform like a smart card.

We overcome these disadvantages by defining a special purpose formal model and customizing state-of-the-art analysis techniques for this model.

1.4 Applications of the Work

In this section we give a high-level description of an application of our formal model and the Polaris tool. We also briefly describe some other possible applications of the work.

1.4.1 Programmable Payment Cards

Our primary application is the aforementioned programmable payment card (PPC). A prototype implementation was created by the OpEm group at Penn. This implementation has been extended to use the policy automata framework described in this dissertation.

Figure 1.1: Smartcards can be as small credit cards (or even smaller).

Payment cards are small plastic cards used in commercial transactions—credit cards (for example, Visa and Mastercard), debit cards (bank ATM cards) and charge cards (American Express) are used widely today. The declining cost and size of computer chips has made it feasible to manufacture 'smartcards'—a card augmented with a small computer. Java Cards are smartcards that have a standard open platform which allows users or other parties to install small applications called applets.

Programmable payment cards exploit this functionality to enforce purchasing restrictions that are more fine-grained than existing credit card systems. Payment cards typically come with a few basic restrictions; a credit card cannot exceed a credit limit, while a debit card cannot spend more money than is in the corresponding bank account. However, a user may want to customize a card to reduce risk or increase convenience. For example, a user may want to temporarily lower the credit limit on a card for budgeting purposes. Such customizations are especially useful when a card is temporarily delegated to someone else. For example, a parent might acquire a credit card and then give it to a child, or an employer will give a company credit card to an employee to cover travel expenses. We refer to parties who pass on these payment cards as *secondary issuers*, and the card issuing company (such as a bank) is referred to as the *primary issuer*. Secondary issuers

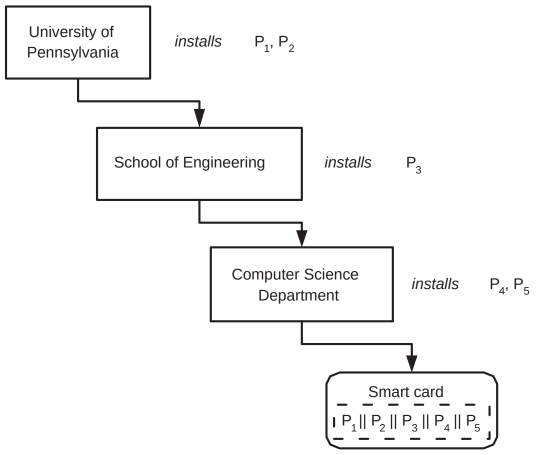

University of Pennsylvania
installs
P_1, P_2
School of Engineering
installs
P_3
Computer Science Department
installs
P_4, P_5
Smart card
$P_1 || P_2 || P_3 || P_4 || P_5$

Figure 1.2: Cascading policies are integrated in one payment card

usually want to enforce some kind of extra restriction; a parent may want a child to only use the credit card for emergencies, while an employer may want to forbid the use of the card for purchasing luxuries.

The programmable payment card allows users to install applets that enforce these restrictions. Before an employer or parent gives the card to the recipient the secondary issuer will install one or more policies on the card. If the recipient attempts to make purchases that violate the policy the card will consult the policies and refuse to authorize the purchase.

A card can store more than one policy. This is useful for a number of reasons. One party's purchasing policy might consist of several independent rules (similar to our swimming pool example) which for the sake of modularity and simplicity are best specified as independent entities. It also allows the card to be used by a hierarchy of secondary issuers, each of which adds one or more policies. For example, consider a card issued to a university, as in Figure 1.2. This card is linked to a particular research grant, which is administered by the university. It would be handy for a computer science professor to use this card to purchase lab equipment for research related to the grant, but various parties want to restrict these purchases to comply with various university, school and departmental policies, as well grant-specific restrictions. Compliance could be monitored after the purchases are made by checking the monthly bill but after-the-fact enforcement is confusing, risky and inefficient. Instead, each party in the hierarchy installs the appropriate policy before passing the card down the hierarchy; the university installs a university-wide policy, the engineering school installs school-wide policies, and so on, until the card is given to the professor. The professor may even delegate it to a graduate student after installing another policy to ensure the student only buys what she has been instructed to buy.

Since these policies are protecting access to a potentially large bank account we would like to have some guarantees about how policies behave and how one policy affects another policy.

We propose using policy automata, described in Chapter 3, to write and analyze these purchase policies.

The programmable payment card is the main application we are considering in this dissertation. It is the only one for which we have implemented a code generator that will translate policy models into actual running programs.

1.4.2 Network Access

We believe our formal model of policies is applicable in domains other than programmable payment cards. The second application we will discuss in this dissertation is network access.

Firewalls are network devices that examine network traffic arriving at or leaving a computer or network. A firewall will be configured with rules that state what traffic should be accepted, forwarded or dropped.

Firewalls are usually configured using vendor-specific configuration files. It is common, however, to have firewall rules formatted as a list where the rules earlier in the list have precedence over the subsequent rules. In practice, managing a firewall is difficult and error prone. When a new rule is added it is difficult to know if the rule is redundant because other rules override it. We believe that our policy model framework can be used to specify rules in a format that is easier to write, understand and manage.

We also think our framework can model and reason about application-specific access policies. Web servers allow administrators to specify who is allowed to access a particular document. We believe that policy automata is an appropriate formalism for representing access to services like web servers.

1.4.3 Other Applications

We think policy models are an appropriate mechanism for encoding policies in many contexts where access control needs to combine multiple policies in an environment where low computational resources are available (whether due to heavy load or restricted hardware or power). For example, a policy model could be used to determine the maximum speed in a car based on the car's recent behavior, the weight being carried and the location of the car. Some work in the University of Pennsylvania Security Lab has examined how our model could be used for cellphones that restrict what kind of calls can be made.

Additionally, policy models could be used to describe policies in many general applications which rely on some form of access control. We envision a developer using Polaris in a manner that is analogous to the way compiler compilers like Yacc [37] are used; when a developer wishes to write an access control module for an application the access control policies will be specified using policy automata which get compiled by Polaris into a general purpose language.

1.5 Use Cases

The core policy automata framework is a general framework which is intended to be suitable for various application domains. In this section we list a few use cases that illustrate how the framework could be used.

1.5.1 Programmable Payment Card

This scenario describes how we envision a developer using the framework to install policies on a programmable payment card.

A developer wants to add a policy to a programmable payment card. The developer uses Polaris to create a set of policy automata that implement her desired policy. She downloads the models for the policies that are already installed on the card. Using Polaris, the developer checks that her new policies will not introduce conflicts with the existing policies or with each other. Some potential conflicts are found so the developer alters her policy automata to avoid the conflict and checks for conflicts again.

Concerned about the limited memory on the card, the developer checks to see if her new policy automata are redundant—in other words, whether her new policy automata actually change the behavior of the card. She discovers that one of her automata is redundant because she accidentally created a trivial policy automaton that never votes to accept or reject a request. She rewrites this automaton so that it behaves correctly. She discovers that another automata is redundant because it duplicates a policy that is already on the card, so she decides not to install the redundant policy.

She then writes some simple test policy automata in order to validate her new policies. These test automata are akin to test scripts or a partial specification—they describe how the policies should behave on a certain classes of input sequences. The developer then checks whether the test policy P_T is redundant with respect to the policies Π for which P_T is a partial specification. If P_T is not redundant then the policies Π do not satisfy the partial specification encoded in P_T and should be rewritten. Fortunately, all the policies the developer wants to add to the policy satisfy the partial specifications.

Once she is satisfied that her policy automata are free of conflicts, useful and have the intended behavior, the developer uses Polaris' code generation feature to generate Java Card applets. Each applet implements one of the new policy automata. These applets are then installed on the card using a standard procedure for adding policy applets. This procedure registers the new policy applets with the existing transaction processing software on the card, which ensures that the policy applets will be consulted for future transaction requests.

1.5.2 Firewall Configuration

This scenario describes how the framework would be used to configure the access policy in a network device.

A firewall administrator creates a set of policy automata, each of which represents one particular concern of the administrator. For example, one automaton guards against a denial-of-service attack, while another ensures that outsiders can send mail to a local mail server.

As in the use case scenario for a programmable payment card, once the administrator has written automata that cover all of the intended rules for admitting or rejecting network packets, she uses the Polaris framework to check for conflicts. She also checks individual policies to see if they are redundant with respect to the other policies. If they are redundant she might remove them (to make the firewall rules shorter and simpler) or re-examine them to see if they really implement the policy she intended. She will also write small test automata that function as partial specifications of the policy, and then verify that her firewall policies meet the partial specifications.

Once she has validated her policy automata she can compile them to a configuration file that, when read by the firewall application, enforces the policies.

If at a later time the administrator wants to modify a policy she can add or remove a particular automaton, re-analyze the new policy automata set, and re-compile the automata into firewall configuration files.

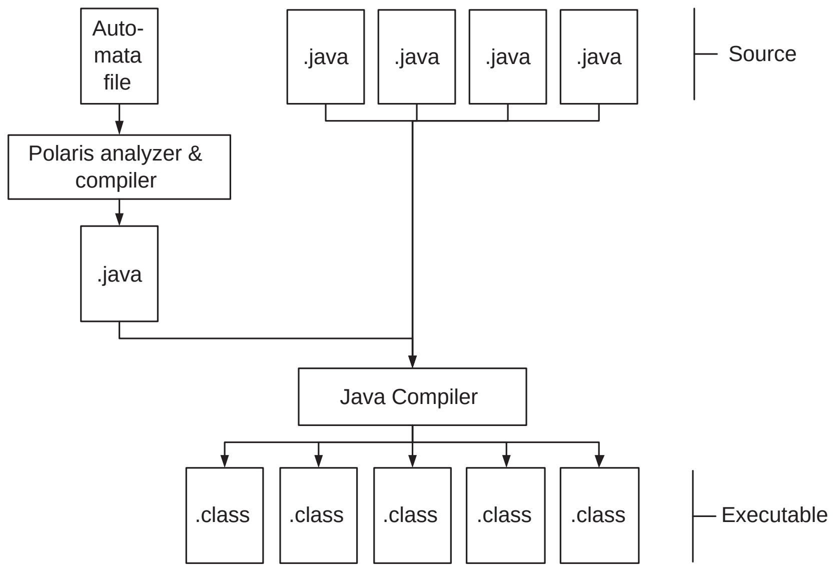


Figure 1.3: Polaris as an access control module compiler

1.5.3 Access Control Module Compiler

This scenario describes how the framework can be integrated in the development process of a general software application that includes some access control functionality. For example, assume a developer is implementing a server based enterprise calendar application which stores users' calendars. Users will be able to view the calendars of other users, but the server must hide particular meetings and other information based on who created the meeting, who was invited, who is viewing the information, and the topic of the meeting.

An access control module that is written in a general purpose programming language will be difficult to write and understand. Since the language is not optimized for expressing policies the programmer will have to worry about low-level implementation details instead of how the policies interact. A general purpose programming language will also be harder to analyze for conflicts or redundancy. Instead, the programmer can use policy automata as a special purpose language for designing an access control policy. The automata will then be compiled into an appropriate general purpose language so that the access control functionality can be integrated with the rest of the application. This compile process is illustrated in Figure 1.3.

This is analogous with the use of tools like *parser generators* (or *compiler compilers*) such as `yacc` [37] or `JavaCC` [35]. If an application needs to incorporate some parsing functionality, a developer will create a file that describes the intended parser using a special syntax (usually the relevant grammar annotated with extra information). This special syntax can be checked for problems specific to parsers, and it can be compiled into a general purpose programming language implementation of the parser.

In our case, a developer creates a set of policy automata using Polaris. The developer performs appropriate validation steps on the automata using the kinds of analysis described in the previous sections—checking for conflicts, redundancy, verifying that the test automata are redundant. Once the developer is satisfied with the results of the analysis, she uses Polaris to compile the automata into a set of Java classes. These classes include an implementation of the policy automata, code to resolve the automata's votes, and interfaces that allow data from the other modules of the application to be conveyed to the policy resolution module.

1.6 Contributions

This dissertation is the first thorough examination of a programmable payment card—a smartcard capable of holding and enforcing multiple modular purchasing policies. Building on the application and architecture developed by the author and others in the OpEm group[22], this dissertation explores the application using a variety of approaches: a formal analysis of the application, an effective language to encode realistic policies, and an implementation of the application as well as a tools for supporting the application.

In our formal examination of the application we proposed *policy automata*, a new formal model of modular access control policies which depend on the history of past transactions. Policy automata resolve conflicts through a voting mechanism which is based on defeasible logic. Information about the transaction history is kept as local state by each policy. Policy automata is a unique formal model of policy integration, and is the first model to combine state machines with defeasible logic.

This model can be seen as an extension of the security automata formalism of Schneider [60] and Ligatti et al.[41]. We have extended this work by defining a new form of enforcement that is appropriate for our application, and proved that suppression automata can enforce exactly the class of safety properties using this form of enforcement. We also extended the work by identifying the problem of building suppression automata through composition—a problem that is solved by composing policy automata which collaborate through our voting mechanism.

Using this model, we have defined formal properties of payment card policies like conflict and redundancy that correspond to useful real-world properties. We have proposed algorithms to check these properties.

In addition to our formal results, we have demonstrated that our formal model can effectively represent real-world purchase policies. In Chapter 4 we show that ten of the twelve purchasing card policies used for the University of Pennsylvania purchase card can be encoded as policy automata. We also demonstrate how our voting mechanism can concisely encode policies which would be cumbersome or impossible with other voting mechanisms.

Finally, we have demonstrated the practicality of our proposed model-based design approach to managing programmable purchase cards by implementing Polaris, a working system for integrating and enforcing payment policies. This implementation enables a policy developer to create policies using an abstract model, convert that model to Java Card applets, and then integrate and run those applets so that they enforce the relevant purchase policies. This implementation includes the first smartcard implementation of a defeasible logic inference algorithm, and we have adapted the known inference algorithm in order to reduce the use of RAM and take advantage of the slower but cheaper EEPROM memory available on the Java Card platform.

Our experimental results show that our system offers acceptable performance for implementing realistic policies, for both a policy designer who wants to analyze or compile policies and a cardholder who wants to purchase items without undue delay during the purchase.

1.7 Structure of the Dissertation

The next chapter gives an overview of the technology and previous research which we have used in this work, as well as other approaches that have been used to solve similar problems. Chapter 3 gives a formal analysis of the application, including a model of security policies. It proposes policy automata as a mechanism for enforcing the policies and describes some formal properties of policy automata. Chapter 4 describes the language we use to easily encode policies and illustrates its expressiveness by showing how a range of realistic policies, including policies drawn from the University of Pennsylvania purchasing rules, can be effectively encoded. Chapter 5 describes our implementation of the Polaris system, which includes an editor, code generator, analysis algorithm and on-card policy management software. Chapter 6 describes some of the security issues raised by our system, including the assumptions we need to consider our system secure. Chapter 7 concludes the dissertation and discusses some open issues and possible future research directions.

Chapter 2

Background

This work builds on an extensive history of research in automata theory, formal methods, model checking, security policies and non-monotonic logic. In this chapter we survey some of the literature that is related to our work.

2.1 Automata Theory

Our policy automata are based on classical finite state systems like finite automata and regular expressions. Literature on finite state systems extends back to the 1940s in work by McCulloch and Pitts [53]. Finite state systems are discussed in standard theory of computation textbooks such as Hopcroft and Ullman's [30], which also includes discussion of composing automata to create new automata (for example, the construction of an automaton that recognizes the intersection of two regular languages). In Section 3.6.1 we compare our formal model to Mealy machines [30], a classical variation of finite automata that writes a sequence of symbols to output instead of simply accepting or rejecting like a finite automaton.

Using state-machine-based models for high-level designs is quite common in software engineering (e.g. Statecharts [25], UML [9]). These models often extend classical finite state automata by adding variables and other high-level language features. Our work on policy automata, especially the Polaris environment for creating automata, is partly inspired by the adoption of these models.

The voting mechanism that we use for composing the decisions by individual policy automata is unusual compared to most languages and formal models of computation, but it is similar to how *combined valued signals* work in the reactive language Esterel [7]. In Esterel, signals are a form of instantaneous output. *Valued* signals contain some data. Since a signal is instantaneous, if two different parts of a program both emit a valued signal there needs to be some way to resolve the two or more signals into a single signal. An Esterel programmer must specify a binary operator like addition, conjunction, or even some programmer-defined operator. If multiple modules emit a signal the actual signal emitted (that is, seen by the environment and other parts of the program) is the result of applying the operator to all the signals. For example, a signal can be declared as:

```
output MySignal := 0: combine integer with +;
```

which indicates that `MySignal` is a signal containing an integer, and multiple signal emissions will be combined using the addition operator. If a program had statements "`emit MySignal(2)`" and "`emit MySignal(3)`" then `MySignal` would take on the value 5. We could also specify a custom operator as follows:

```
output MyOtherSignal
      := 0: combine integer with CustomOp;
```

where `CustomOp` is an arbitrary binary operator implemented in another language.

2.2 Formal Methods and Model Checking

This work builds on a wide range of previous work in formal methods [16], especially in model-checking [15] techniques. One example of a mature model checking tool is SPIN [29], which explores the reachable states of a system by performing a depth-first-search of the execution paths of the system.

Hermes [2, 3] uses a language in which state-machines are extended with scoped variables, exceptions, data structures and code re-use. A system is specified using a hierarchical graphical language. Hermes has enumerative and symbolic state search algorithms which are optimized to exploit the hierarchical structure of a system. For example, if a procedure is called from multiple locations in the system then, when searching the possible execution paths, Hermes will attempt to re-use information about the procedure. If the procedure was called in one context then subsequent calls from other contexts will not trigger another search of the procedure. Hermes also conserves memory by ignoring state information that is not relevant at a given program location—for example, if a variable is out of scope at a location in the program then the memory used to track that variable can be used for other data. Polaris uses much of Hermes' code for manipulating, saving and type-checking state-machine-based languages.

2.2.1 Models with Logic Extensions

There is some formal methods work which combines non-traditional logics and state-machine-based models. Easterbrook and Chechik [14] analyze merged state machines by using paraconsistent logics to capture the possibly inconsistent views of the system. Siddiqi and Atlee [61] use a hybrid model that combines state-transitions and logical assertions to model and analyze feature interaction conflicts in telephone systems. Hay and Atlee [26] define composition operators that allow labeled transition systems to execute in parallel without conflicts, possibly by overriding the effects of low-priority transitions. Neither approaches are obviously suitable for modeling and analyzing the policies of the type we model using policy automata.

2.2.2 Formal Methods for Java

This work was partly inspired by the need to reason about the behavior of policy applets that were written for the OpEm Programmable Purchase Card project [22]. We have chosen a model-based approach in which we use a high level model to describe the policy applet's behavior and rely on automated tools to generate executable code from the model. An alternate approach would be to check the behavior of the applets using a formal methods tool for Java. Even with our model-based approach there is a possible role for such tools to check properties of imported functions—we discuss this issue in Section 4.1.2.

The Java Modeling Language (JML) is a standard for annotating Java source with special comments that express properties of the code. Tools like the LOOP compiler [70], the Extended Static Checker for Java (ESC/Java) [18] and ESC/Java(2) [32] can check that the code actually satisfies the properties specified in JML. [12] gives an overview of JML and its tools.

The Bandera project [68] develops tools for validating Java programs by writing specifications and then verifying those specifications using model-checking and static analysis techniques. NASA's Java Pathfinder tool [71] uses model-checking to find runtime errors like uncaught exceptions, deadlocks and violated assertions.

2.3 Non-monotonic Logic

A *non-monotonic logic* is an extension of traditional logic that models the non-monotonic reasoning that is common in the real world. In traditional logic we make conclusions based on known facts. If new information is added to the system the conclusions that we have already made are still valid. Traditional logic is therefore monotonic; new facts can only lead to new conclusions. In non-monotonic logic new information may force us to retract conclusions. In Chapter 1 we have already mentioned a standard example: most birds fly, so if we are told that Tweety is a bird then we can tentatively conclude that Tweety can fly. However, if we later learn that Tweety is a penguin we will be forced to retract our conclusion that Tweety flies.

The family of non-monotonic logics contains many different formalisms of non-monotonic reasoning. Brewka et al. give an overview of the various approaches in [11]. Here we mention the formalisms that are most related to our voting system.

At the end of the 1970s a number of non-monotonic reasoning systems were first proposed. Reiter [59] proposed *default logic*, which extends traditional logic by extending the known facts of a theory with a set of defaults of the form

$$\frac{A : B_1, \ldots, B_n}{C}$$

where $A, B_1, \ldots, B_n, C$ are all classical formulas. The default is interpreted as follows: if A is provable and $\neg B_i$ is not provable for all $i = 1, \ldots, n$ then we can conclude C. For example, we can write a default expressing the notion that birds can usually fly as

$$\frac{\mathrm{bird}(x) : \mathrm{flies}(x)}{\mathrm{flies}(x)}$$

In plain English, this default says that we can conclude that a given bird can fly unless we have evidence that it cannot fly (that is, unless $\neg\mathrm{flies}(x)$ is provable). We can write a more specific default as

$$\frac{\mathrm{bird}(x) : \mathrm{flies}(x), \mathrm{hasWings}(x)}{\mathrm{flies}(x)}$$

This states states that if x is a bird and we have no reason to believe the x cannot fly, and we have no evidence that the x has no wings, then we can conclude x flies.

In a default logic theory the set of facts is extended by applying defaults to generate new conclusions. These new conclusions may then make other defaults applicable. Applying defaults until a the set of facts reaches a fixed point gives us an *extension*. The inferences of a theory are those formulas which are contained in all possible extensions.

A similar non-monotonic formalism is *maximal consistency logic* [58]. In maximal consistency logic, classical logic premises are sorted by priority. For example, we can write our running example as

$$\begin{aligned} p_1 &: \quad \mathrm{bird}(x) \Rightarrow \mathrm{flies}(x) \\ p_2 &: \quad \mathrm{penguin}(x) \Rightarrow \neg\mathrm{flies}(x) \\ p_3 &: \quad \mathrm{bird}(x) \wedge \mathrm{penguin}(x) \end{aligned}$$

where $p_3 > p_2 > p_1$. Since $p_2 > p_1$ we should 'prefer' conclusions that rely on p_2 over those that rely on p_1.

Both default logic and maximal consistency logic are expressive enough to describe a variety of complex policies where some sub-policies defer to other higher priority sub-policies. Unfortunately, computing inference for these logics involves computation intensive operations like computing fixed-points and choosing among many chains of reasonings (for example, checking inference in default logic is not even in NP [13]). This makes them undesirable for resource-constrained devices like smart cards and many network devices. It also makes analysis more difficult.

To work around this problem of intractability we choose Nute's *defeasible logic* [56], which is a pared-down non-monotonic logic that is designed for efficiency. It differs from the approaches described above in that it only contains literals and inference rules about literals. This makes inference much easier to compute; [47] gives a linear time algorithm. A detailed introduction to defeasible logic is given in Section 3.3.2.

Non-monotonic logics often give paradoxical or non-intuitive results. This is especially undesirable for use in modeling security policies, where a mistake in the policy can lead to security breaches. To mitigate this problem we isolate the defeasible logic from the state update operations of our policy by restricting the logic to the voting system.

2.4 Policy Languages

Various policy specification languages have been proposed. Damianou et al. [17] use the Ponder language to describe access control policies. Hoagland et al. [28] use a graphical language to describe security policies. Both of these approaches target a wide range of access control applications and it is not clear how amenable the languages are to analysis.

Lupu and Sloman [42] discuss a number of strategies for resolving policy conflicts, including assigning explicit priorities to policies, choosing policies that are 'closer' to the subject of the policy, or defaulting to denying permission. For each choice they give examples where the strategy is problematic.

There is related work using non-monotonic logics for reasoning about policies. Grosof et al. [21] represent business rules using courteous logic programs, while Antoniou et al. [4] use defeasible logic to represent administrative regulations governing, for example, exam scheduling. These approaches encode the entire policy as statements of non-monotonic logic statements; in contrast, we isolate the logic part of our model in the voting mechanism, and update state and choose votes using pure state-machine mechanisms. As discussed in the previous section, this separation is motivated by the desire to use a formalism where it is most appropriate; defeasible logic is effective at resolving conflicts, while state machines are effective at recording state. We were concerned that encoding a policy entirely in defeasible logic would increase the chance of design errors, since non-monotonic logics are prone to paradoxes and are unfamiliar even to many programmers, let alone people who design policies. This separation also allows us to treat the voting mechanism as a parameter in our framework—if another voting mechanism is preferred much of the formal results and implementation described in this dissertation would still be applicable.

Miro [27] uses a graphical language, allows policies to override other policies, and analyzes policies, but it is targeted at file system security.

Halpern and Weissman [24] propose using a fragment of first-order logic called Lithium as a security policy model which accommodates merged policies and has a tractable algorithm to determine access rights. The restrictions that ensure tractability guarantee that the merged policies are consistent. Like our policy model formalism, Lithium is designed to be efficient and handle policy composition. Lithium assumes an environment of facts that are available to the policy engine, while our policy automata update their own state to record the information relevant to their decisions. We chose this strategy primarily because we wanted to avoid storing extraneous data about past transactions since the smart card platform has so little memory available. We were also motivated by a desire to preserve a very simple interface between a policy enforcer program and whatever system manages transactions—this interface would be more complex if policies had to query this system to make policy decisions.

Stoller and Liu [63] propose a technique that takes a trust management policy described in Datalog and generates a lightweight implementation that checks the policy, allowing the use of such policies in resource-constrained contexts like embedded systems. A security policy framework has two components: a policy language and an algorithm which, given a policy and a request for service (for example, a request to enter a building), checks whether the request satisfies the policy. One can choose a policy language so that there is an algorithm with which any request can be checked against any policy efficiently. Instead, Stoller and Liu suggest optimizing the algorithm for a specific policy (which presumably changes only infrequently) so that the policy-specific algorithm can check requests efficiently.

2.5 Security Automata

Schneider [60] uses *security automata* to model access control policies and generate monitors that enforce correct behavior. Policies are treated as predicates on sets of traces, and Schneider identifies a subset of policies which can be enforced by automata-based runtime monitors.

Ligatti et al. [41] extend this work by generalizing Schneider's automata to include automata which block bad actions or fill in missing actions. They show how these new automata differ from Schneider's automata with respect to the formal definition of policy used in [60].

The policy automata formalism proposed in this dissertation can be seen as an adaptation and extension of this line of research; our policy models are effectively the same as the automata which block forbidden actions—in our case, undesirable transactions. We adapt this line of work by showing how the formal definitions effectively model a concrete application—the programmable purchase card. We give an effective method for composing automata, since the composition technique proposed in [60] does not generalize to the automata of [41]. Such a technique is important because policies are much easier to understand and modify if they can be broken into smaller sub-policies; any policy enforcement mechanism that can handle the large lists of policies that a real enterprise require will need to be able to compose policies. Sections 3.1 and 3.2 discuss the relevant aspects of the security automata work in detail.

Fong [19] classifies security automata by the amount of state they keep and examines how such limits impact the policies they can enforce.

2.6 Java Card

Java Card [54] is a standard open platform designed to run on a smartcard. A Java Card compliant smartcard has a small virtual machine which runs applets which are written in a version of Java [5] adapted for low-resource environments.

The platform is described in specifications [64, 66, 65] that are available for free online. These specifications leave open many of the details of dynamically installing applets. The GlobalPlatform standard [20] fills in many of the details regarding the management of multiple applications on a single smart card.

Lyubich [45, 43, 44, 46] has implemented the Secure Electronic Transaction (SET) protocol [49, 50, 51, 52], a secure purchase protocol, on a Java Card. This software has been extended by the OpEm group at University of Pennsylvania [22] to implement a prototype of the programmable payment card application, where the user can install applets that approve or reject transactions before the transaction takes place. We extend this OpEm implementation for some of our experimental results in Chapter 5.

The Java Card designers recognized the dangers of letting users install arbitrary applets. A Java Card virtual machine (VM) must enforce *applet firewalls*, a mechanism for preventing objects from one applet manipulating objects in other applets. Java Card applets are also required to respect the Java's strong typing scheme, but verifying applets are well-typed has traditionally been performed off-card, since the verification has been assumed to require too much memory. Leroy [39] has proposed a modified applet format and algorithm that can be executed on card, removing the need to trust the compiler that generated an applet.

2.6.1 Formal Analysis Work on Java Cards

In recent years, there has been a lot of research on formal methods for Java cards, especially by the VerifiCard project [34]. Much of this work uses the JML tools mentioned in [12]. This research typically focuses on proving correctness of protocols and API implementation, or ensuring that applets behave as specified [10]. To our knowledge, the problem of adding policies dynamically and merging them with existing policies has not been addressed beyond verifying that an applet respects the constraints of the Java Card platform.

2.7 Network Access Policies

Guttman's *filtering postures* work [23] and the Firmato tool [6] of Bartal et al. use domain specific high-level languages to describe firewall policies for a set of networked computers. The languages are specific to firewall rules and do not describe stateful policies that react to the history of arriving packets. These approaches are also directed at distributed policies, while our approach focuses on the policies of a single device. While Firmato does not perform any formal analysis on the models of firewalls, Guttman presents an algorithm for checking that the distributed firewall implementation satisfies a high level policy. Neither approach treats policies as independent modules with priorities that change according to circumstance.

We think it is possible that these approaches could be combined with our formalism.

In such an approach policies would be described using policy automata instead of simple pattern matching on packets (for example, packets going to port 80 at address 1.2.3.4), while the algorithms and language for managing multiple network devices would remain mostly unchanged.

In addition to Guttman's work, there are several tools that analyze firewall rules. For example, Wool's Lumeta firewall analyzer [72] generates a list of all traffic that a firewall permits, and highlights common firewall configuration errors. This tool only analyzes static firewall configurations and can not therefore handle the stateful policies we model with our formal framework.

Chapter 3

Formal Framework

In this chapter we analyze the programmable payment card application formally. We begin by introducing the *security automata* work of Schneider[60] as a basis for a formal understanding what a payment card policy is. We go on to discuss what it means to enforce such policies, introducing a new notion of enforcement appropriate for our application. We show how *suppression automata*, introduced by Ligatti et. al[41], are capable of enforcing a class of policies, but lack an effective composition mechanism that would allow a policy designer to construct complex enforcement mechanisms from simpler mechanisms. To solve this problem, we introduce a new formal model called *policy automata*, which combines state machines with a voting mechanism based on defeasible logic to effectively and succinctly model payment card policies. We discuss the semantics of the policy automata model and define some properties of policy automata that would be of interest to a policy designer. We also present some algorithms and techniques for checking that a set of policy automata satisfy such properties. Finally, we characterize the expressiveness of our new model by discussing what can and cannot be modeled, and by comparing the model to classical models of computation.

3.1 General Policies

Schneider[60] investigated the properties of formal definitions of security policies. This work concentrated on policies that can be enforced by a run-time monitor—for example, a monitor that wraps untrusted mobile code so that it cannot harm the environment in which it is being executed. Such a monitor could watch a program and block any attempt to send information on a network after a disk has been read. Schneider proposed *security automata* as a formal model of enforcement mechanisms. This work was later extended by Ligatti et al.[41], who examined automata with extra capabilities. The security automata framework is general enough to be applied to our programmable payment card application where we want to protect financial resources like a bank account from a user is only partially trusted.

In this section we present the aspects of the security automata framework that are relevant to our application, adapting the examples and properties to our particular application. We follow the presentation in [41].

3.1.1 Security Policies

Let T be a finite set of events. We write T^* for the set of finite-length sequences of events in T. A *security policy* is a predicate on sets of event traces. In other words, a set of traces $\Sigma \subset T^*$ satisfies security policy P if and only if $P(\Sigma)$.

In our case, the set of events is the set of possible transactions a user attempts to make. For example, an event could be "buy one gallon of paint from Home Depot for \$30 at 11am on November 2, 2004".

We use the following notation for sequences: We write the empty sequence as $\cdot$, and use the notation $\sigma; \tau$ to denote the concatenation of sequences σ and τ. We write $\sigma[..i]$ for the i-length prefix of σ, and $\sigma[i..]$ for the sequence that includes the i-th element of σ and all subsequent elements. Therefore $\sigma = \sigma[..i]; \sigma[(i+1)..]$.

The set of security policies defined above is broad enough to capture policies like non-interference: given two events $a, b \in T$, we may require that the appearance of an event a

yields no information about whether event b appears in a trace. In other words, the policy is true for a set $\Sigma \subset T^*$ if b never appears in a trace in Σ or some traces have both a and b while others have b without a. In the context of mobile code b may be an event visible to outsiders while a occurs when the last bit of a secret key is 1. If the policy holds then an observer who sees b cannot infer anything about the value of the secret key. In the context of payment cards such a policy could be used to prevent corruption. Event a may denote the cardholder receiving a payment (for example, an election campaign contribution) from a merchant, while b denotes the cardholder buying a large item from the merchant. The payment a may be an innocuous event or it may be a bribe to secure the cardholder's future business. If the event b is independent of a then we can be confident that a was not a bribe.

Another policy could enforce a credit limit on a credit card. Let the event a_i denote spending $\$i$ on an item where i could range from 1 to 100. A policy to enforce a $50 credit limit would have the following predicate:

$$P = \{\sigma \in T^*. \sum_{a_i \in \sigma} i < 50\}$$

3.1.2 Policy Classes

Alpern and Schneider[1] identify a subset of security policies which are called *properties*. A property is a policy that can be identified by looking at each execution trace without referring to other possible traces. Formally, a policy P is a *property* if there is a predicate $\hat{P}$ over traces such that

$$P(\Sigma) = \forall \sigma \in \Sigma. \hat{P}(\sigma) \tag{3.1}$$

Note that our non-interference policy is not a property; we can not tell if the sequence $a; b$ is permitted without checking to see if there is another sequence with a b but no a in Σ. (The non-interference policy needs to reason about different possible traces and is therefore reminiscent of branching-time temporal logics, which can reason about the existence of different execution paths starting from a common state. In contrast, a property necessarily focuses on a single trace and is therefore reminiscent of linear-time temporal logics. In fact, the term *property* comes from the literature on linear-time concurrent program verification.) A predicate $\hat{P}$ on traces induces a property P on sets of traces so when we refer to $\hat{P}$ as a property we mean the induced property P that satisfies (3.1). Our credit limit policy is a property, as the total money spent in one trace does not depend on the other traces accepted by the predicate.

A *safety property* is a property $\hat{P}$ such that

$$\forall \sigma \in T^* [\neg \hat{P}(\sigma) \Rightarrow \forall \tau \in T^* . \neg \hat{P}(\sigma ; \tau)] \tag{3.2}$$

There are properties that are not safety properties. For example, a policy may require that every time a cardholder borrows money—event b—the cardholder eventually pays the money back—event p. This policy is a property; one can check that a given trace obeys the policy without examining other traces. However, the sequence b violates the policy (the money is borrowed without being paid back) while the sequence $b; p$ obeys the property. If we set $\sigma = b$ and $\tau = p$ then we see that (3.2) is not satisfied. Intuitively, a safety property is a property that can be verified by looking at a finite prefix of the trace—we do not have to wait to see if some event occurs later in the trace which will change a bad trace into a permitted trace.

3.1.3 Enforcing Policies

Schneider introduced security automata as a mechanism for enforcing policies. These automata read a series of events emitted by a target program and if they detect a policy violation they terminate the target program. Ligatti et al. generalized these automata, defining additional automata classes which can remove events, insert events, or do both. The first of Ligatti's classes, called *suppression automata*, is of special interest to this work, as it is an appropriate model for a payment card monitor.

Formally, a security automaton is a deterministic finite or countably infinite state machine (Q, q_0, Δ). Q is the set of states of the machine. The initial state is q_0. The *transition function* Δ specifies how the automaton reacts to its input—Δ varies for different types of automata, and will be specified in detail below. The automaton reacts to input in a series of steps of the form $(\sigma, q) \xrightarrow{\tau} (\sigma', q')$ where σ is the sequence of events that the target wishes to execute and q is the state of the automaton before the step is taken; σ' is the sequence of events waiting to be executed after the step and q' is the state of the automaton after the step; $\tau \in T^*$ is a sequence of events which take place during the step. Only the events in τ are observable—these are the only events which actually impact the environment. For example, in a programmable payment card, the observable events are those transactions which take place. We write $(\sigma, q) \overset{\tau}{\Rightarrow} (\sigma', q')$ to denote zero or more steps yielding output τ.

Ligatti [41] identifies two abstract principles for effectively enforcing a property:

Soundness An enforcement mechanism must ensure that all observable outputs obey the property.

Transparency An enforcement mechanism must preserve the semantics of executions that already obey the property in question.

These principles take a fairly liberal view of enforcing policies. Consider an enforcement mechanism which outputs no events for an input that violates a policy. Such a enforcement mechanism would satisfy both principles, but such draconian enforcement would not be satisfactory in a programmable payment card context, where a cardholder who accidentally violates a policy would like to be able to continue making valid purchases. We therefore add the following (imprecise) principle:

Minimality An enforcement mechanism must ensure that observable outputs differ from the input as little as possible.

This principle will be made precise below.

Ligatti et al. formalize the soundness and transparency principles as follows:

Definition: An automaton A with starting state q_0 *precisely enforces* a property $\hat{P}$ on the system with event set T if and only if $\forall \sigma \in T^*, \exists q' \in Q \; \exists \sigma' \in T^*$ such that

1. $(\sigma, q_0) \overset{\sigma'}{\Rightarrow} (\cdot, q')$,

2. $\hat{P}(\sigma')$, and

3. $\hat{P}(\sigma) \Rightarrow \forall i\ \exists q''.\ (\sigma, q_0) \overset{\sigma[..i]}{\Rightarrow} (\sigma[i+1..], q'')$

An automaton that precisely enforces a property will accept any input sequence that satisfies the property and output the same sequence of events. Furthermore, if a property does not satisfy a property the automaton will output a sequence of events that does satisfy the property. Additionally, the automaton works in lockstep with the target on a valid input: every time the automaton reads an input event it outputs the same event.

Precise enforcement ensures soundness (only valid output sequences are produced) and transparency (a valid input sequence will simply be copied to output). However, precise enforcement puts no restrictions on the behavior of an automaton given an invalid input sequence. Assume $T = \{a, b\}$ and assume that property $\hat{P}_b$ requires that event b never occurs. Consider the automaton A_b with the following behavior:

- $(a; \sigma, q_0) \xrightarrow{a} (\sigma, q_0)$
- $(b; \sigma, q_0) \xrightarrow{\cdot} (\cdot, q_0)$

A_b terminates as soon as the first b is seen. If no b events are seen then the automaton simply copies the input a event to output. The automaton therefore precisely enforces $\hat{P}_b$. However, given sequence $abaa$ the automaton will only output a and then halt, when outputting aaa would satisfy the property and be, in some sense, closer to the target's intended execution trace. The automaton does not satisfy the minimality principle.

3.1.4 Suppression Automata

Ligatti et al. examine a family of automata that can enforce policies. They classify automata into four families: *truncation automata*, *suppression automata*, *insertion automata* and *edit automata*. As only the first two are required in this discussion we will ignore insertion automata and edit automata. Interested readers can see the details in [41].

A *truncation automaton* is a security automaton specified by (Q, q_0, δ) where, as in the case for security automata, Q is a finite or countable set of states, and q_0 is the initial state. The transition function is a partial function $\delta : T \times Q \rightarrow Q$ that specifies how the automaton reacts to input. The truncation automaton updates its state as follows:

- $(a; \sigma, q) \xrightarrow{a} (\sigma, q')$ if $\delta(a, q) = q'$
- $(\sigma, q) \xrightarrow{\cdot} (\cdot, q)$ otherwise.

The automaton copies input events to output so long as δ is defined on the current state and input event. When a state and input event is reached for which δ is not defined the automaton halts, terminating the target.

While a truncation automaton enforces a policy by terminating a target, a *suppression automaton* enforces a policy by blocking certain actions. More formally, a suppression automaton has four components (Q, q_0, δ, ω), where Q and q_0 have the same definition as in a truncation automaton, and δ is again a partial function $\delta : T \times Q \rightarrow Q$. The partial function $\omega : T \times Q \rightarrow \{0, 1\}$ specifies whether an input event should be copied to output (1) or suppressed (0). The possible single steps of a suppression automata are:

- $(a; \sigma, q) \xrightarrow{a} (\sigma, q')$ if $\delta(a, q) = q'$ and $\omega(a, q) = 1$.
- $(a; \sigma, q) \xrightarrow{\cdot} (\sigma, q')$ if $\delta(a, q) = q'$ and $\omega(a, q) = 0$.
- $(a; \sigma, q) \xrightarrow{\cdot} (\cdot, q)$ otherwise.

Ligatti et al. showed that truncation automata can precisely enforce a property if and only if the policy is a safety property. Perhaps surprisingly, the ability of a suppression automaton to selectively block events rather than halt execution does not allow suppression automata to precisely enforce any more properties than truncation automata. A suppression automaton can also precisely enforce a property if and only if the property is a safety property[1].

[1]Ligatti et al. show that if we allow an automaton to change a valid input sequence into a different

Recall our property $\hat{P}_b$ which requires that event b never be observed. If the monitoring target is a piece of untrusted mobile code then it may be reasonable to halt the target when the forbidden event is encountered. However, if the monitoring target is a programmable payment card holder and event b denotes a purchase of a forbidden item it seems natural to let the cardholder continue making purchases even though an attempt was made (perhaps accidentally) to violate the purchase policy. For example, the University of Pennsylvania purchasing policy says that (among other restrictions) box lunches can be bought with the corporate card but bottled water cannot. Given that it would simple to forget small details of such a complex policy, blocking all purchases after any violation seems too draconian. A suppression automaton has the option of permitting purchases after denying a purchase. Let S_b be the suppression automaton with the following possible steps:

- $(a; \sigma, q) \xrightarrow{a} (\sigma, q)$
- $(b; \sigma, q) \xrightarrow{\cdot} (\sigma, q)$

On input $abaa$ our truncation automaton A_b simply outputs a. The suppression automaton S_b outputs aaa—a sequence much 'closer' to the input sequence. In some sense, this makes suppression automata a more powerful class of security automata, since, in addition to soundness and transparency, they can satisfy our minimality principle. We make this precise below.

We would like a formal definition of a type of enforcement that captures this sense that suppression automata are a 'better' enforcement mechanism than truncation automata. We do so using an abstract partial order, and then investigate two candidate partial orders.

An *approval sequence* is a finite-length binary sequence $\beta \in \{0,1\}^*$. We define an operator $\otimes : T^n \times \{0,1\}^n \rightarrow T^*$ that removes events in a sequence when the corresponding

but semantically equivalent output sequence then suppression automata are more powerful than truncation automata. If we were willing to let our cards behave more as agents who are permitted to split a single transaction into multiple transactions, switch a purchase from one merchant to another similar merchant, or otherwise manipulate transactions, then semantic equivalence of transactions output sequences would applicable to programmable payment cards. However, we suspect users would uncomfortable with such functionality and would allow cards to perform a very minimal amount of manipulation at most. In this work we therefore concentrate on automata which are limited to blocking transaction requests.

element in the approval sequence is a 0. Formally,

$$(a;\sigma) \otimes (1;\beta) = a; (\sigma \otimes \beta)$$
$$(a;\sigma) \otimes (0;\beta) = (\sigma \otimes \beta)$$

For example, $abaab \otimes 10011 = aab$, $abc \otimes 011 = bc$. Essentially, the $\otimes$ operator selects a subsequence of an event sequence. Note that $\sigma \otimes 1^{|\sigma|} = \sigma$.

Given a partial ordering $\prec$ of approval sequences, an automaton A *$\prec$-gracefully enforces* a property $\hat{P}$ if and only if

- A precisely enforces $\hat{P}$, and
- if $(\sigma, q_0) \overset{\sigma'}{\Rightarrow} (\cdot, q')$ then $\exists \beta \in \{0,1\}^*.\ \sigma \otimes \beta = \sigma' \wedge \hat{P}(\sigma \otimes \beta') \Rightarrow \beta' \not\prec \beta$

(Note that β and β' are necessarily the same length in the above definition.) Informally, an automaton gracefully enforces a property if it precisely enforces a property and if the input sequence violates the property, the automaton suppresses just enough events to make the input sequence valid. If there is some other subsequence of events that can be removed to make the event sequence valid, then that subsequence is no smaller (according to our partial order) than the subsequence chosen by the automaton. We can view this as a target's desired sequence of events degrades gracefully when it violates a property.

One obvious candidate for a partial order is to order approval sequences by the number of 0's in the sequences; intuitively, the less rejected events the better the automaton. Let $\prec_{\#}$ be the partial order such that $\beta \prec_{\#} \beta'$ if and only if β contains fewer 0s than β'. Note that 1^n is the minimal sequence of length n, which corresponds to our intuitive notion that we want to approve everything we can safely approve. However, there are simple properties which cannot be $\prec_{\#}$-gracefully enforced by a suppression automaton.

Claim 1 *There is a property that cannot be $\prec_{\#}$-gracefully enforced by a suppression automaton.*

Proof. Consider a set of events $T = \{1, 2, 3\}$ representing the number of dollars spent in a transaction and a property which requires that no more than \$4 be spent in total. Assume A is a suppression automaton that $\prec_\#$-gracefully enforces the property.

The event sequence 1;3 does not violate the property, so since A precisely enforces the property A must simply copy the input events to output. The first 2 steps of A's execution on 1;3 are $(1; 3, q_0) \xrightarrow{1} (3, q_1) \xrightarrow{3} (\cdot, q_2)$.

If cardholder attempts to make the following sequence of purchases 1;3;1;1 then the minimal possible approval sequence (using the $\prec_\#$ ordering) is 1;0;1;1. However, A's next step depends entirely on the current state and the next input event. So the first two steps of A's execution will be identical to the steps listed above: $(1; 3; \sigma, q_0) \xrightarrow{1} (3; \sigma, q_1) \xrightarrow{3} (\sigma, q_2)$. Which means A does not induce the 1;0;1;1 approval sequence, which therefore means A does not $\prec_\#$-gracefully enforce the property. ∎

Obviously the $\prec_\#$ partial order does not yield a reasonable definition of graceful enforcement; it requires an automaton to select a minimal but valid approval sequence that may depend on values of the input sequence which are not immediately available—something which is impossible for some properties. There are other more practical reasons to object to such enforcement; a cardholder who tries to purchase an item that will not break his credit limit would probably be upset if his card rejected the purchase in order to approve two or more later purchases.

We define a more practical partial order $\prec_l$ which uses an ordering similar to lexicographic ordering. The order is defined as follows:

- $1 \prec_l 0$
- $1; \sigma \prec_l 0; \sigma'$
- $\sigma \prec_l \sigma' \Rightarrow 1; \sigma \prec_l 1; \sigma' \wedge 0; \sigma \prec_l 0; \sigma'$

Informally, approving the first event is more minimal than rejecting the first event. If two approval streams agree on the first event then we order them by whatever order is implied by comparing the streams without the first element. Once again, 1^n is the minimal sequence of length n.

Theorem 2 *Any safety property $\hat{P}$ has a corresponding suppression automaton that can $\prec_l$-gracefully enforce $\hat{P}$.*

Proof. The proof extends the proof of Ligatti et al. [41] that shows that every safety property can be precisely enforced by a truncation automaton.

We construct a suppression automaton that $\prec_l$-gracefully enforces $\hat{P}$ as follows:

- States: $q \in T^*$ (the sequence of events seen so far). To distinguish between sequences and states representing sequences we write $\overline{\sigma}$ for the state representing sequence σ.

- Start state: $q_0 = \bar{\cdot}$ (the state representing the empty sequence)

- Transition function δ and the approval function ω: In state $\overline{\sigma}$ if we see input event a then

  – If $\hat{P}(\sigma; a)$ then $\omega(a, \overline{\sigma}) = 1$ and $\delta(a, \overline{\sigma}) = \overline{\sigma; a}$

  – If $\neg\hat{P}(\sigma; a)$ then $\omega(a, \overline{\sigma}) = 0$ and $\delta(a, \overline{\sigma}) = \overline{\sigma}$

Note that it always holds that if the state of the automaton is $\overline{\sigma}$ then σ is the output sequence seen so far and $\hat{P}(\sigma)$. This can be shown by an induction on the steps of the automaton.

We need to consider two cases to show that the automaton $\prec_l$-gracefully enforces the property $\hat{P}$ for any sequence $\sigma \in T^*$.

- **Case** $\hat{P}(\sigma)$: When $\hat{P}(\sigma)$ the automaton behaves exactly as the automaton constructed in the proof of Theorem 1 in [41]. So we know that the automaton precisely enforces $\hat{P}$ on σ. Let $\beta = 1^{|\sigma|}$, the minimal approval sequence of length $|\sigma|$. The automaton emits $\sigma \otimes \beta = \sigma$ and, since β is the minimal sequence possible, $\hat{P}(\sigma \otimes \beta') \Rightarrow \beta' \not\prec_l \beta$ for any other approval sequence β' of lentgth $|\sigma|$. So the automaton $\prec_l$-gracefully enforces $\hat{P}$.

- **Case** $\neg\hat{P}(\sigma)$: Let σ' be the event sequence emitted by the automaton. It's clear that $\hat{P}(\sigma')$ since the automaton can only emit valid event sequences. Let β be the approval sequence induced by the suppression automaton. We have $\sigma \otimes \beta = \sigma'$ and $\hat{P}(\sigma')$. Let β' be an approval sequence such that $\hat{P}(\sigma \otimes \beta')$ and $\beta' \prec_l \beta$. Using the definition of the $\prec_l$ ordering, we can find $x, y, y' \in \{0,1\}^*$ such that $\beta = x;0;y$ and $\beta' = x;1;y'$ (x may be an empty sequence). Recall that we write $\sigma[..i]$ for the i-length prefix of σ. In this case, let $i = |x|$, the length of x. The automaton will have emitted $\sigma[..i] \otimes x$ after i steps of the automaton running on input σ. Therefore the automaton will be in state $\overline{(\sigma[..i] \otimes x)}$. Since $\beta = x;0;y$ it follows that $\omega(\sigma[i+1], \overline{(\sigma[..i] \otimes x)}) = 0$ so it must have been the case that $\neg\hat{P}(\sigma[..i+1] \otimes (x;1))$. However, $\hat{P}$ is a safety property, so by (3.2), $\forall\tau \in T^*.\neg\hat{P}((\sigma[..i+1] \otimes (x;1));\tau)$. Note that $\sigma[..i+1] \otimes (x;1)$ is a prefix of $\sigma \otimes (x;1;y')$ and $\beta' = (x;1;y')$ so $\neg\hat{P}(\sigma \otimes \beta')$, which contradicts our assumption about β'. So no such β' can exist and the β induced by the suppression automaton is minimal. $\square$

The $\prec_l$ partial order therefore gives us a practical notion of graceful enforcement; any property that can be precisely enforced by a suppression automaton can also be $\prec_l$-gracefully enforced. Using this notion of enforcement we see that suppression automata are more powerful than truncation automata since, as discussed in Section 3.1.4, a truncation automaton cannot $\prec_l$-gracefully enforce any policy where dropping mid-sequence events yields a valid event sequence.

3.1.5 Reject-Blind Automata

A suppression automaton (Q, q_0, δ, ω) is *reject-blind* if whenever $\omega(a,q) = 0$ then $\delta(a,q) = q$. Informally, a *reject-blind* automaton does not record rejections. A suppression automaton which is not reject-blind is a *reject-observing* automaton.

The suppression automaton constructed in the proof of Theorem 2 is reject-blind. This means that we do not need the full class of suppression automata to $\prec_l$-gracefully enforce all safety properties—the class of reject-blind suppression automata is sufficient.

However, any automaton that $\prec_l$-gracefully enforces a property must by definition precisely enforce that property. Ligatti et al. showed that suppression automaton can only precisely enforce safety properties. Therefore, the class of reject-observing automata cannot $\prec_l$-gracefully enforce any property that cannot be $\prec_l$-gracefully enforced by a reject-blind automaton. In some sense, the smaller class of reject-blind automata is as powerful as the full class of all suppression automata.

Reject-observable automata *are* capable of behaviors that cannot reproduced by reject-blind automata. However, this behavior cannot be observed in the output trace of the automaton, and therefore it cannot be distinguished by the security automata framework, which classifies policies by their sets of acceptable output traces.

For example, many automated teller machines will disable a bank card (by taking away the card) if the cardholder cannot type the correct PIN within three attempts. Consider a suppression automaton A_{atm} with a similar policy: block all bad events b and if there are three b's in a succession then block all further events. This automaton is easy to encode; we just increment a counter every time we reject a b, reset the counter when we see a non-b event, and if we see a b with the counter showing two previous b's we enter a state where all further events are rejected. This automaton is necessarily reject-observing, as we need to update our state when we reject a b. However, the set of output traces generated by this automaton is exactly the set of traces allowed by an automaton which simply drops b events without ever disabling the card. For example, on input $a; a; b; b; b; a$ the automaton A_{atm} will output $a; a$, which is also the result of simply dropping the b events from the trace $a; b; a; b; b$ (among other traces). Any sequence σ of non-b events is a possible output of A_{atm} since we get σ as output if we feed $\sigma; b; b; b$ to A_{atm}. Therefore the ATM policy cannot be expressed using the formal definition of security policy from Section 3.1.1.

3.2 Composition

The security automata framework of [41] examines the capabilities of a complete monitor automaton which enforces one property. This work does not address how multiple properties or automata should be composed. Schneider [60] proposed composing multiple truncation automata[2] by taking a simple conjunction of the automata; when one of the automata wants to truncate the target the combined automata truncate the target. The resulting property is the conjunction of all the constituent properties; the combined automata accept the intersection of all the traces accepted by the constituent automata.

This ability to enforce a policy as a composition is desirable for several reasons, as discussed in Section 1.1. It allows distinct policies to be described in isolation for simplicity and clarity. However, conjunction seems to be less appropriate for suppression automata than truncation automata, as it is not clear if taking a conjunction of the constituent ω functions yields the desired result. Consider a property $\hat{P}_{ab}$ of traces of $T = \{a, b, c\}$ which insists that event b can only occur immediately after an a. An automaton $A_{ab} = (Q_{ab}, q_0, \delta_{ab}, \omega_{ab})$ could enforce such a property by setting ω_{ab} to reject any b unless it was preceded by an a. Given the invalid sequence $a; b; b; c$ as input, the automaton would emit the valid sequence $a; b; c$. Consider another property, $\hat{P}_{\neg a}$ which disallows traces which begin with a. An automaton $A_{\neg a} = (Q_{\neg a}, q_0', \delta_{\neg a}, \omega_{\neg a})$ could enforce this property by setting $\omega_{\neg a}$ to reject event a if it is seen in the initial state. After the first non-a event the automaton moves to a state where all events are accepted. Given input $a; b; c$ the automaton will output the sequence $b; c$.

What happens when we take the conjunction of these automata and feed it the event sequence $a; b; c$? The automaton A_{ab} will accept the first event since its ω_{ab} function yields 1, but the automaton $A_{\neg a}$ will reject the first event. Taking the conjunction of the ω functions we reject the first event. Automaton A_{ab} will then accept the b event because, from its viewpoint, the b occurs after an a—the automaton has no mechanism to record the fact that another automaton has rejected an event. Our composed automaton will eventually

[2]In [60] the term *security automata* is used solely for what Ligatti et al. call truncation automata.

emit $b;c$ given input $a;b;c$ even though such an output trace violates property $\hat{P}_{ab}$. Unlike for truncation automata, a conjunction of suppression automata does not enforce the conjunction of the individual properties.

We could construct a suppression automaton covering a number of properties $\hat{P}_1, .., \hat{P}_n$ by taking the conjunction of all the properties. In other words, set $\hat{P}(\sigma) \Leftrightarrow \bigwedge_{i=1}^{n} \hat{P}_n(\sigma)$ and construct an automaton manually or using the construction described in the proof of Theorem 2. However, this is unsatisfactory from an engineering point of view; the resulting property may be very complex and difficult to encode, and the suppression automaton construction algorithm in the proof of Theorem 2 may not yield a concise automaton. A preferable solution would allow us to run a set of automata in parallel, as we can with truncation automata.

As discussed in Section 1.1, there are situations where, even if we had some system to combine suppression automata in a well-behaved conjunction, we would want a more subtle method to combine policies. We may want a policy that can override other policies; for example, a policy allowing lifeguards to enter a pool should override a policy barring swimmers from a pool after business hours. A conjunction of these two policies will deny the lifeguard access to the pool.

From this discussion we derive two requirements for a mechanism for composing suppression automata: first, automata need to be able to react to the approval/disapproval decision to properly update their state, and second, automata should be able to submit a variety of possible opinions on whether to accept or reject, including the option of deferring to or overriding other automata.

Our solution is to extend the definition of a suppression automaton to allow it to record rejections of events by other automata with which it has been composed. We extend the range of the ω function so that instead of yielding 1 or 0, it yields an element in a set D of *votes*. To avoid confusion, we use $\gamma : Q \times T \to D$ to refer to this extended version of ω. We add a *resolution function* $f : 2^D \to \{\mathsf{yes}, \mathsf{no}, \top\}$ which combines the votes from individual automata into a yes or no or $\top$, indicating approval, rejection or conflict.

Finally, we extend the domain of the transition function δ so that it includes the approval or disapproval as a parameter. This model of composable policy enforcement is described in the next section.

3.3 Encoding Policies

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*.

3.3.1 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 $\{\mathsf{yes}, \mathsf{no}, \top\}$, representing *accept*, *reject* and *conflict* (or error) respectively. The meanings of accept and reject are the obvious ones. A *conflict* result signifies that the votes offer conflicting opinions about whether to accept or reject a transaction request.

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 it contains no and does not contain yes; and to $\top$ if it contains both a yes and a no or only maybe.

For a more complex example, we can model votes with varying priorities if we set D to be the set of integers. We interpret integer $n > 0$ as a vote for acceptance while we interpret $n < 0$ as a vote to reject. The absolute value of the vote indicates the priority of the vote, where higher values have higher priority. For example, if the votes were -5 and 3 then they would be resolved as no or reject, since the reject vote has the higher priority. Any set of votes where the accept and reject votes had the same maximum absolute value would yield a conflict. A set with no votes or with only 0 would also yield conflict.

Another resolution strategy for the same set of votes would be to take the sum of all votes, with a positive, negative and zero sum yielding yes, no and conflict, respectively.

At times we will treat D and $f : 2^D \rightarrow \{\mathsf{yes}, \mathsf{no}, \top\}$ as abstract mathematical entities. We need an actual D and f, however, for the implementation of our framework and to get a concrete sense of how policy models capture real world policies.

We have the following requirements for D and f:

Expressive: The votes should be rich enough to allow various ways of combining and prioritizing different policies.

Succinct: The votes should succinctly express the policy. They should be easy to write and maintain. The votes of a single policy should not have to be re-written when that policy is composed with a new policy.

Efficient: There should be an efficient algorithm for evaluating f on a set of votes. This is especially important in applications for devices with limited computational power. An efficient algorithm for f will also make analyzing the policy model more feasible.

Well understood: An ideal system of votes would be based an a system that has been studied in the literature previously instead of something that we invent.

For our payment card application we use *defeasible logic* to describe and resolve votes. As we show in the next section, defeasible logic is rich enough to express various ways of combining votes. It is succinct enough to express tentative preferences without explicitly ranking votes. At the same time, there is an efficient algorithm to compute f. Finally, defeasible logic comes off-the-shelf—it was invented in the 1980s and has been studied as a purely logical system and as a way to model real-world regulations [55, 56, 48, 47, 8].

3.3.2 Defeasible Logic

In this section we introduce defeasible logic, following the presentation of [47]. Readers who want a more detailed explanation and discussion of the logic are referred to [56, 47].

Atomic formulas and their negations make up the *literals* of defeasible logic. For example, from atomic formulas p, q we get four literals: $p, q, \neg p, \neg q$. The *complement* of a literal l is written $\sim l$; the complement of an atomic formula p is $\neg p$ and the complement of a negated atomic formula $\neg p$ is p. In other words, if p is an atomic formula then $\sim p = \neg p$ and $\sim (\neg p) = p$.

Defeasible logic has three kinds of *rules*:

Strict rules Strict rules are like normal implication:

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

The meaning of this rule is "if $penguin$ is true then *fly* is not true".

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

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

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

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

will block a rule like $bird \Rightarrow \mathit{fly}$ since the knowledge that a bird is injured counters 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. For example, in the rule

$$\mathit{fly}, \mathit{mammal}, \mathit{scary} \Rightarrow \mathit{bat}$$

we tentatively conclude bat only if fly, mammal and scary are all true. If a rule has an empty set of literals on the left hand side then we write the left hand side as "$\{\}$", as in "$\{\} \Rightarrow q$".

The literals on the left hand side of a rule are the *antecedents* of the rule. We denote the antecedents of a rule r as $A(r)$. The literal on right hand side of the rule is the *consequent* of the rule, and we use $C(r)$ to refer to the consequent of a rule r.

We can assign priorities to rules by giving a partial ordering of rules. This ordering determines which rule to apply when two rules conflict. For example, if we have two rules

$$\begin{aligned} r_1 : \quad & \mathit{injured} \Rightarrow \neg \mathit{strong} \\ r_2 : \quad & \mathit{big} \Rightarrow \mathit{strong} \end{aligned}$$

then if $r_1 < r_2$ we will conclude that an elephant that is big and injured is strong; rule r_1, which suggests that such an elephant is not strong, is overridden by r_2 since r_2 is superior to r_1 in the ordering.

Inference in Defeasible Logic

A defeasible logic *theory* consists of a set F of *facts* (literals known to be true), rules R, and a partial order relation $>$ on R. Given a theory we can construct a *derivation* using the inference rules for defeasible logic. A derivation is a sequence $P = P(1), \ldots, P(n)$ of *tagged literals,* literals annotated with a tag indicating what we have proved about the literal. In defeasible logic there are two notions of provability and each form of provability has a positive and negative tag. The four possible tagged literals corresponding to a literal l are:

- $+\Delta l$: l has been definitely proved. Informally, l has been proved using strict rules and facts.
- $-\Delta l$: l cannot be definitely proved. Informally, we have shown that $+\Delta l$ will never be derived.
- $+\partial l$: l is defeasibly provable. Informally, l has been proved using both defeasible and strict rules, in addition to facts.
- $-\partial l$: l cannot be defeasible proved. We have shown that $+\partial l$ will never be derived.

We use the following notation for various subsets of R, a set of rules. The set of strict rules is denoted by R_s. We use R_{sd} for the set of strict and defeasible rules, R_d for the set of defeasible rules, R_{dd} for the set of defeasible and defeater rules, and R_{dft} for the set of defeater rules. We write $R[q]$ for the set of rules with consequent q. This notation extends to subsets of R so that, for example, $R_d[q]$ is the set of defeasible rules with consequent q.

The derivation sequence $P = P(1), \ldots, P(n)$ is constructed incrementally. Each step in the construction adds one element $P(i+1)$ to P based on the elements $P(1), \ldots, P(i)$ and one of four inference rules which are described below. Each rule corresponds to one of the four types of tagged literals described above. We write $P(1..i)$ to refer to the sequence $P(1), \ldots, P(i)$.

The first two inference rules deal with definite provability. This is provability in the classical monotonic sense. We make conclusions based on chains of implication without worrying if some other chain of implication contradicts our conclusion.

Rule $+\Delta$: We can append $P(i+1) = +\Delta q$ if either

$q \in F$ or

$\exists r \in R_s[q].\, \forall a \in A(r).\ + \Delta a \in P(1..i)$

We can mark a literal q as definitely provable if it is a fact or it can be proved using a strict rule where the antecedents of the rule are all definitely provable.

For example, if our theory has one fact a and one rule $a \rightarrow b$ (where both a and b are atomic formulas) then we can apply rule Rule $+\Delta$ once for the first step of the derivation $P(1) = +\Delta a$ (since a is in F) and once again for the second step $P(2) = +\Delta b$ since there exists a strict rule implying b with all antecedents (that is, a) tagged as definitely provable.

The rule for marking a literal as impossible to prove definitely has a similar structure to the previous inference rule:

Rule $-\Delta$: We can append $P(i+1) = -\Delta q$ if

$q \notin F$ and

$\forall r \in R_s[q].\ \exists a \in A(r).\ -\Delta a \in P(1..i)$

We can mark a literal q as definitely unprovable if it is not a fact and all the rules that can strictly conclude q are disabled because they depend on literals which cannot be definitely proved.

For example, if our theory has no facts and one rule $a \rightarrow b$ then we can apply Rule $-\Delta$ to get derivation step $P(1) = -\Delta a$ because $R_s[a]$ is empty (no rules imply a), and then we can apply the rule again to get step $P(2) = -\Delta b$ since the only rule in $R_s[b]$ is $a \rightarrow b$ which has an antecedent a which has been shown to be impossible to definitely prove.

The next two inference rules deal with defeasible provability, for which we must consider competing chains of implication.

Rule $+\partial$: We can append $P(i+1) = +\partial q$ if either

(1) $+\Delta q \in P(1..i)$ or

(2) (2.1) $\exists r \in R_{sd}[q].\forall a \in A(r).\ +\partial a \in P(1..i)$ and

(2.2) $-\Delta \sim q \in P(1..i)$ and

(2.3) $\forall s \in R[\sim q]$ either

(2.3.1) $\exists a \in A(s).-\partial a \in P(1..i)$ or

(2.3.2) $\exists t \in R_{sd}[q]$ such that

$\forall a \in A(t).\ +\partial a \in P(1..i)$ and $t > s$

If a literal q is definitely provable then it is defeasibly provable; if clause (1) is true then we can apply Rule $+\partial$. Otherwise, we need to show (2) that there is rule which implies q which is not overruled by a competing rule. Clause (2.1) ensures that the rule implies q and its antecedents are defeasibly provable. (2.2) checks that $\sim q$, the complement of q, has been shown to be unprovable. (2.3) checks that any rule implying $\sim q$ is either inapplicable because they depend on antecedents that are not provable (2.3.1) or they are overridden by a rule implying q that has a higher priority in the ordering of rules (2.3.2).

Consider a theory with one fact, $bird$, and one rule $r_1 : bird \Rightarrow \mathit{flies}$. We can apply inference rule $+\Delta$ to get $P(1) = +\Delta bird$ (since $bird$ is a fact). We can then apply inference rule $+\partial$ for $bird$ since clause (1) above applies, yielding $P(2) = +\partial bird$. We can apply inference rule $-\Delta$ to get $P(3) = -\Delta\neg \mathit{flies}$ since there are no rules implying $\neg \mathit{flies}$. Finally, we can apply inference rule $+\partial$ again for flies to get $P(4) = +\partial \mathit{flies}$ since clause (2) applies: there is a defeasible rule implying flies with all antecedents marked as defeasibly provable (2.1), and we have shown that $\neg \mathit{flies}$ cannot be definitely proved, and there are no rules implying $\neg \mathit{flies}$ (2.3). Note that we have shown that flies is defeasibly provable without showing it to be definitely provable. In fact, it is impossible to generate a derivation for $+\Delta \mathit{flies}$ in this theory.

Consider the same theory above with an additional fact $injured$ and an additional rule $r_2 : injured \rightsquigarrow \neg \mathit{flies}$ with no ordering on r_1, r_2. We can take the same inference steps described in the preceding paragraph except for the last step. In that step, clause (2.3.1) does not hold because there is a rule $r_2 \in R[\neg \mathit{flies}]$ whose antecedent is not tagged as impossible to prove defeasibly. In fact, adding this additional fact and rule makes it impossible to have a derivation containing $+\partial \mathit{flies}$.

The rule for marking a literal q as impossible to defeasibly prove is similar in structure to the preceding rule:

Rule $-\partial$: We can append $P(i+1) = -\partial q$ if

(1) $-\Delta q \in P(1..i)$ and

(2) (2.1) $\forall r \in R_{sd}[q].\exists a \in A(r). -\partial a \in P(1..i)$ or

(2.2) $+\Delta \sim q \in P(1..i)$ or

(2.3) $\exists s \in R[\sim q]$ such that

(2.3.1) $\forall a \in A(s). + \partial a \in P(1..i)$ and

(2.3.2) $\forall t \in R_{sd}[q]$ either

$\exists a \in A(t).\ -\partial a \in P(1..i)$ or $t \not> s$

In order to mark q as not defeasibly provable we need to check that it is not definitely provable (1) and that defeasible implications are impossible (2). Showing that defeasible implications are impossible requires us to show that (2.1) all rules implying q are blocked because one of their antecedents is not provable, or (2.2) that the complement of q has been shown to be definitely provable, or (2.3) that there is a rule that implies $\sim q$ that is enabled (2.3.1) and is not overruled by a competing rule with a higher priority (2.3.2).

Consider a theory with no facts and one rule $b \Rightarrow a$. We can apply inference rule $-\Delta$ to get $P(1) = -\Delta b$ since there are no strict rules implying b and b is not a fact. Similarly, we can apply the same inference rule to get the derivation step $P(2) = -\Delta a$ since there are no strict rules implying a (the rule $b \Rightarrow a$ is defeasible, not strict). We can then apply the inference rule $-\partial$ for the literal b because clause (1) holds for b and clause (2.1) holds since the set $R_{sd}[b]$ is empty. This gives us $P(3) = -\partial b$. Once we have shown b to be impossible to prove defeasibly we can apply the inference rule $-\partial$ again, this time for the literal a. The derivation step $P(2)$ gives us clause (1) and while $R_{sd}[a]$ is not empty since it contains the rule $b \Rightarrow a$, the antecedent b in the rule has been tagged as impossible to prove defeasibly, so (2.1) holds.

We say that a tagged literal tl is a *conclusion* of a theory $(F, R, >)$ if we can apply the inference rules described above to yield a derivation where $P(i) = tl$ for some i. We denote this as $(F, R, >) \vdash tl$.

3.3.3 Defeasible Logic as a Voting Mechanism

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 AF which is fixed for an application. The atomic formula yes is one element of AF. Let $\mathcal{R}$ be the set all rules (strict, defeasible and defeater) made of elements of AF. 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.

For our voting mechanism we set F, the set of facts, to be empty. To state that a literal is true we can include a rule with an empty set of antecedents. For example, $\{\} \rightarrow a$ will imply that a is provable, essentially making a a fact. We also assume that the $>$ order on rules is trivial, in the sense that no rule is greater than any other rule. These restrictions simplify the voting mechanism and they also let us optimize the inference algorithm—for example, clause (2.3.2) in inference rules $+\partial$ and $-\partial$ becomes trivial. We have found that even with the restrictions mentioned above the voting mechanism is expressive enough to encode the policies we want to encode. (If extra flexibility is needed it would be fairly simple to extend the formal framework and implementation to handle facts and rule orderings.) With these restrictions the defeasible logic theory is entirely determined by the set V of votes (which gives a set of rules), so we write $V \vdash tl$ to state that a theory made from votes V yields conclusion tl.

The resolution function f on argument $V \subset D$ is defined as follows:

- $f(V) = \mathsf{yes}$ if $V \vdash +\partial\mathsf{yes}$ and $V \not\vdash +\partial\neg\mathsf{yes}$.
- $f(V) = \mathsf{no}$ if $V \not\vdash +\partial\mathsf{yes}$.
- $f(V) = \top$ if $V \vdash +\partial\mathsf{yes}$ and $V \vdash +\partial\neg\mathsf{yes}$.

Note that it is possible for both yes and $\neg\mathsf{yes}$ to be defeasibly provable in defeasible logic.

Consider the following three votes:

$$\begin{aligned} v_1 :&\quad \{\} \Rightarrow p; q \Rightarrow \texttt{yes} \\ v_2 :&\quad p \rightarrow q \\ v_3 :&\quad \{\} \rightarrow \neg\texttt{yes} \end{aligned}$$

Evaluating f on these votes gives us $f(\{v_1, v_2, v_3\}) = \textsf{no}$ since v_3 concludes (without preconditions) that `yes` is not defeasibly provable. However, if we only consider the first two votes then $f(\{v_1, v_2\}) = \textsf{yes}$ since the v_1 allows us to tentatively conclude p, v_2 allows us to conclude q (given p), and the second rule of v_1 allows us to conclude `yes` given q.

3.3.4 Other Voting Mechanisms

The primary voting mechanism we investigate in this work is the defeasible logic voting mechanism described in the previous section. However, for illustration and comparison we will occasionally employ other voting mechanisms. We describe four different mechanisms in this section.

Definition: In the *binary voting mechanism* the set of votes D_2 is just $\{\textsf{true}, \textsf{false}\}$, indicating approval and disapproval respectively. The voting function simple takes the conjunction of all the votes: $f_2(V) = \textsf{yes}$ if $\bigwedge_{v \in V} v$, otherwise $f_2(V) = \textsf{no}$. $\square$

Note that conflicts are impossible in this voting mechanism. The binary voting mechanism is very simple, and is essentially the same as using conjunction to compose suppression automata as discussed in Section 3.2.

Definition: The *3-valued logic voting mechanism* has a set of votes $D_3 = \{\textsf{true}, \textsf{false}, \perp\}$, where $\textsf{true}$ indicates approval, $\textsf{false}$ indicates rejection, and $\perp$ indicates that we have no preference. The resolution function f_3 is evaluated on votes $V \subset D_3$ as follows

- $f_3(V) = \textsf{yes}$ if $\textsf{true} \in V$ and $\textsf{false} \notin V$.

- $f_3(V) = \textsf{no}$ if $\textsf{false} \in V$ and $\textsf{true} \notin V$.
- $f_3(V) = \top$ if $\textsf{true}, \textsf{false} \in V$ or $\textsf{true}, \textsf{false} \notin V$.

□

The next voting mechanism resembles the majority voting of political elections.

Definition: In the *election voting mechanism* the set of votes D_e is $\{\textsf{true}_i, \textsf{false}_i, \bot_i\}$, the values of three valued logic tagged with unique identifier[3], and the resolution function $f_e(V)$ returns yes if the true votes outnumber the false votes, no if the false votes outnumber the true votes, and $\top$ if the the true and false votes are equally numerous. □

Our next voting mechanism is a generalization of the 3-valued logic mechanism where we can annotate true and false with priority levels.

Definition: The *prioritized logic voting mechanism* has a set of votes

$$D_p = \{\bot\} \cup \{\textsf{true}, \textsf{false}\} \times \{1, 2, \ldots\}$$

A vote of $\bot$ indicates no preference, a vote of $(\textsf{true}, n)$ is a vote to approve with priority n, and a vote of $(\textsf{false}, n)$ is a vote to reject with priority n. If a vote $(\textsf{true}, n)$ conflicts with a vote $(\textsf{false}, n')$ then the vote with the higher priority takes precedence. More formally, we define a function pri where $\text{pri}(\bot) = 0$ and $\text{pri}(b, n) = n$. Let $\max(V) \subset V \subset D_p$ be the set of votes with the maximal priority. In other words, $\max(V) = \{v \in V.\ u \in V \Rightarrow \text{pri}(v) \geq \text{pri}(u)\}$.

- $f_p(V) = \textsf{yes}$ if $\max(V) = \{(\textsf{true}, n)\}$ for some n.
- $f_p(V) = \textsf{no}$ if $\max(V) = \{(\textsf{false}, n)\}$ for some n.
- $f_p(V) = \top$ if $V = \emptyset$ or $\max(V) = \{\bot\}$ or $\max(V) = \{(\textsf{false}, n), (\textsf{true}, n)\}$.

□

[3]We tag the votes so that when we take the union of all votes we retain information about the number of votes for true, false and $\bot$.

We will refer to the components of the defeasible logic voting mechanism as simply D and f. Other voting mechanisms will be identified with a subscript: D_3 and f_3, and D_p and f_p, etc.

3.3.5 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.

Definition: A *policy automaton* P is a tuple (Q, q_0, γ, δ). The components of P are

Q A set of *states*

q_0 An initial state

γ The *voting function* of P. γ is a function

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

which determines how the policy automaton votes in a given state to process a given transaction.

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. $\square$

Note that the transition function δ is not defined for votes in which the resolution function returns $\top$; as we see below, if the resolution function yields $\top$ the set of automata enters a special error state.

As we discuss in the Chapter 4, in practice the policy automaton is specified using a graphical language. We split the automaton state into *modes* (similar to control points in a program) and *variables*. The modes are expressed as vertices in our graphical language. 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 vote statements that refer to the current state and the transaction parameters, and specify the function γ.

Definition: 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 $\{\mathsf{yes}, \mathsf{no}, \top\}$. $\square$

Since D and f are usually, respectively, the subsets of the set of defeasible logic rules and the function defined in Section 3.3.3, we will sometimes conflate a policy model (Π, D, f) and the underlying set Π of policy automata. For example, when we write a model $M = M' \cup \{P\}$ we mean $M = (\Pi \cup \{P\}, D, f)$ where $M' = (\Pi, D, f)$.

Consider the following payment card policy: "Allow at most one purchase over \$100. All purchases $\leq \$100$ are allowed unless a purchase is made for $\geq \$200$, after which no purchases will be allowed at any price." Assume transactions t consist of a single value representing the price (for example, $t = 25$). A policy automaton that implements this policy could be described as follows. We let $Q = \{q_0, q_1, q_2\}$, where state q_0 is the initial state where we have seen no purchases over \$100, q_1 is the state after one purchase of over \$100, and q_2 is the state after one purchase of over \$200. The function γ maps states to votes as follows:

$$\begin{array}{lcl} (q_0, t) & \mapsto & d_{\mathsf{yes}},\ \forall t \\ (q_1, t) & \mapsto & d_{\mathsf{yes}},\ \forall t \leq 100 \\ (q_1, t) & \mapsto & d_{\mathsf{no}},\ \forall t > 100 \\ (q_2, t) & \mapsto & d_{\mathsf{no}},\ \forall t \end{array}$$

where d_{yes} is a single defeasible logic rule $(\{\} \rightarrow \mathsf{yes})$ which forces the literal yes to be provable (that is, forces the request to be accepted) and d_{no} is the opposite rule $(\{\} \rightarrow$

$\neg\textsf{yes})$ that forces a rejection. The function δ updates state as follows:

$$\begin{array}{rcl} (q_0, t, \textsf{yes}) & \mapsto & q_1 \ \text{ for } 200 \geq t > 100 \\ (q, t, \textsf{yes}) & \mapsto & q_2 \ \text{ for } t \geq 200, \forall q \in Q \\ (q, t, -) & \mapsto & q \ \text{ otherwise} \end{array}$$

where the '$-$' in the last line indicates that the mapping applies whether transaction request was approved or not. In the initial state q_0 all purchases are approved. The transition function switches states from q_0 to q_1 when a purchase of > 100 is made, thereby disallowing further purchases > 100. If the purchase is ≥ 200 then the transition switches to state q_2, thereby preventing any future purchases.

3.3.6 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 $f(\vec{d}) =$ 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 $\delta(q, t, \textsf{yes})$. Similarly, if the transaction request t is rejected, the state will be updated to be $\delta(q, t, \textsf{no})$.

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 $a = \textsf{yes}$ or $a = \textsf{no}$, 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

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

When it is not clear which policy model we are referring to we will subscript the update notation with the model. For example, for policy model M we will write $q \overset{t\uparrow a}{\Longrightarrow}_M q'$ and $q \overset{\tau\uparrow\alpha}{\Longrightarrow}_M q'$.

3.4 Properties of Policy Automata

In this section we define and investigate some interesting properties of policy automata.

3.4.1 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.

A simple example of a conflict with the defeasible logic voting mechanism is a combination of two votes

- $v_1 : \{\} \rightarrow \neg\mathsf{yes}$
- $v_2 : \{\} \rightarrow \mathsf{yes}$

Both yes and ¬yes are asserted to be true, which makes both literals provable, forcing a conflict. This conflict can be avoided if one of the automata defers to the other; for example, vote v_1 could be changed to "$\{\} \Rightarrow \neg\mathsf{yes}$", which only asserts yes if other votes do not contradict the vote.

3.4.2 Redundancy

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

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

The definition of redundancy above only applies to a policy automaton's behavior when combined with a given set of automata. However, in a situation where a policy developer expects additional policies to be installed on the card this redundancy may not be appropriate. Consider a policy automaton P which has a single vote, which is always enabled for all transaction requests, of the form $a \rightarrow yes$. This vote forces an accept when the literal a is true. If this policy is combined with a policy model M with no automata with votes concluding a (that is, no votes with implication rules with a on the right hand side of the rule) then the implication $a \rightarrow yes$ will never be triggered, so this vote will never affect the approval of a transaction request. Obviously, P is redundant in M. We then add a policy P_a which has a vote of the form $\{\} \rightarrow a$, which asserts that a is true, to M to make M'. The vote of P will be activated since a is now true, so P will now affect what gets emitted by the automaton. In other words, P is redundant in M but not $M' = M \cup \{P_a\}$. We can strengthen the notion of redundancy so that a policy is redundant no matter what policies are installed in the future.

Definition: Given a policy model $M = (\Pi, D, f)$ where $\Pi = \{P_1, \ldots, P_k\}$, policy automaton P is *strongly redundant in* M if for all finite sets Ψ of policy automata, and for all sequences τ of transaction requests, $M' = (\Pi \cup \Psi, D, f)$ emits α on τ if and only if $M' = (\Pi \cup \{P\} \cup \Psi, D, f)$ emits α on τ. □

It is easy to see that strong redundancy implies redundancy; if P is strongly redundant in A then it is redundant for $A \cup \Psi$ where Ψ is the empty set. The converse is not true, as we showed in the example above where P was inactive until P_a was added to the model.

What kind of policies are strongly redundant? If a policy automaton has only empty votes—votes which consist of zero defeasible logic rules—then it will not ever affect the inference algorithm, so it will be redundant in all models, and therefore strongly redundant in all models. If a policy automaton P only contributes redundant votes to a model—in other words, whenever P gives vote v there is an automaton in the model giving the same vote v—then P will never affect the outcome of the inference algorithm; such a P will be strongly redundant in that model.

More generally, we define a *redundancy ordering* to be a partial order $\preceq$ of D such that

$$d \succeq d' \Rightarrow \forall V \subset D. f(V \cup \{d, d'\}) = f(V \cup \{d\}) \tag{3.3}$$

If an automaton P's vote d is always dominated in the $\preceq$ ordering by one of the votes in an automaton in a policy model M, then P is strongly redundant in M.

Example 3 *If we choose the set of prioritized logic votes (D_p, f_p) for our voting mechanism, we can order votes according to their priorities: $v \preceq_p u \Leftrightarrow \mathrm{pri}(v) \leq \mathrm{pri}(u)$. Since f_p ignores any votes that have less than maximal priority, our $\preceq_p$ ordering satisfies 3.3. So if an automaton P always supplies a vote v that is of lower priority than a vote contributed by an automaton in a model M, then P is strongly redundant in M.*

A similar ordering exists for our defeasible logic voting mechanism. Intuitively, an implication rule $a_1, a_2, a_3 \rightarrow c$ is redundant if there is already a rule $a_1, a_2 \rightarrow c$ present; whenever the first rule is triggered the second rule will also be triggered, so if the first rule is dropped the conclusions that can be inferred from the rules will not change. A careful case by case analysis of the defeasible logic inference algorithm confirms this intuition: if a defeasible logic theory contains rules $r_1 : a_1, .., a_n \rhd c$ and $r_2 : a_1, .., a_i \rhd c$ where $i < n$ and $\rhd$ is one of $\rightarrow, \Rightarrow, \leadsto$, then we can drop r_1 without changing the tagged literals that can be derived from the rules. This gives us a partial order on votes satisfying (3.3). Recall that we use $A(r)$ to denote the antecedents of the rule r and $C(r)$ to denote the consequent of the rule r. We get the following order:

$$v \preceq_r u \Leftrightarrow \forall r \in v.\, \exists s \in u.\, C(s) = C(r) \wedge A(s) \subseteq A(r)$$

A policy automaton P that always submits votes that are smaller, using the $\preceq_r$ ordering, than one or more votes from the automata in a model M is therefore strongly redundant in M.

3.4.3 Refinement

Refinement is a concept that has been studied in the context of formal models of computation. Informally, a program (or an agent, or a module, etc.) p refines p' if it is safe to replace p' with p. It is often the case that a refinement relation can make program analysis easier; we can check a simple program and infer that a more complex refinement of the simple program behaves similarly.

One natural definition of refinement in the context of event sequences is to use a subset relation as the refinement relation; if a suppression automaton A emits a set of event sequences Σ then a suppression automaton A' which enforces $\Sigma' \subseteq \Sigma$ can safely replace A since anything that A disallows will be disallowed by A'. For truncation automata there is a natural correspondence between refinement and composition. If a truncation automaton A_1 precisely enforces a policy which admits a set Σ_1 of traces, and another truncation automaton A_2 precisely enforces a policy that allows the set Σ_2 of traces, then composing the two automata yields an automaton which allows $\Sigma_1 \cap \Sigma_2$. A composition of automata is therefore a refinement of each of the constituent automata. This suggests the following refinement relation:

Definition: A policy model (or suppression automaton) M with output traces Σ_M is a *sequence refinement* of a policy model N with output traces Σ_N if and only if $\Sigma_M \subseteq \Sigma_N$. We write this as $M \leq_s N$. $\square$

As noted above, if we compose two truncation automata A_1 and A_2 to make a truncation automaton A_3, then $A_3 \leq_s A_1$ and $A_3 \leq_s A_2$.

This correspondence between composition and intersection does not hold for policy automata in general; a counter-example is described below. However, if we use the binary voting mechanism (D_2, f_2), where votes are either true or false and votes are resolved by taking the conjunction of all votes, then composition does correspond to intersection if the policy automata are reject-blind.

We defined reject-blindness for suppression automata in Section 3.1.5. We extend the definition for policy automata and policy models in the natural manner: a reject-blind policy automaton is a policy automaton (Q, q_0, γ, δ) where $\delta(q, t, \mathsf{no}) = q$ for all $q \in Q, t \in T$. Informally, a reject-blind automaton does not record transaction requests which are rejected. It is simple to see that a policy model that is constructed from reject-blind policy automata will be a reject-blind suppression automaton.

If a policy model M_1 allows output sequences $\Sigma_1 \subseteq T$ and policy model M_2 allows output sequences $\Sigma_2 \subseteq T$ then, assuming both models use the (D_2, f_2) voting mechanism, if we compose the models by taking the union of their policy automata then the composed policy model M_3 will only allow output sequences in $\Sigma_1 \cap \Sigma_2$, so $M_3 \leq_s M_1$ and $M_3 \leq_s M_2$.

However, if we allow automata which can update state after rejecting a transaction request then the composition of two policy models may allow output events which are not allowed by one of the models in isolation. The following example illustrates such a case.

Example 4 *Let $T = \{a, b, c\}$. A policy automaton $P_1 = (Q_1, q_{init1}, \gamma_1, \delta_1)$ that only accepts sequences of the form $a; b; c; a; b; c; a; \ldots$ could be encoded with three states, $q_a (= q_{init1}), q_b$ and q_c, with q_i indicating that only transaction request i will be accepted. When reading input $a; b; c; a; b; c; a; \ldots$ the automaton will move from from $q_a \rightarrow q_b \rightarrow q_c \rightarrow q_a$ and so on. We can set the transition function δ_1 so that this state update happens even if the transaction request is rejected by another automaton. Note that P_1 is not reject-blind since it switches states even when a transaction request is rejected. In isolation, the automaton P_1 will only accept traces of the form $a; b; c; a; b; c; a; \ldots$*

Consider another automaton P_2 which accepts all transaction until one c transaction *request has been approved. After this point, P_2 accepts a's and b's but not c's. If we compose P_1 and P_2 by combining them in a policy model the resulting model will behave as follows on the input sequence $a; b; c; a; b; c; a$. The automaton P_2 will vote to reject the second c transaction request, while P_1 will vote to accept all the events (since P_1 updates its state whether or not the event was rejected by P_2). The resulting output sequence will be $a; b; c; a; b; a$, which is not an output sequence admitted by P_1 in isolation.*

This example shows how even a simple voting mechanism coupled with automata that record rejections can lead to behavior where composing automata does not lead to a sequence refinement of the individual policy automata. The following refinement relation, priority refinement, attempts to capture the sense that a more specific or precise policy automaton is safe replacement for a less specific policy automaton. Intuitively, a policy model N refines M if N's decision differs from M only when N's decision is more definite (or has a higher priority) than M's decision.

Definition: *Priority refinement:* Let C be a partially ordered set. We will call this set the *consolidated vote set*. We order sequences in C^* by individually comparing the elements of the of the stream; given sequences $s = c_1, c_2, \ldots$ and $s' = c'_1, c'_2, \ldots$, we say $s \leq s'$ only if $c_i \leq c'_i$ for all i.

Let g be a function from subsets of D (the set of votes) to C. Given a transaction request sequence σ and a policy model M, let $s(M, \sigma)$ be the sequence $c_1, c_2, \ldots$ generated by taking letting $c_i = g(\vec{d_i})$ where $\vec{d_i}$ are the votes that M yields on the i-th transaction of σ. If $c < c'$ for $c, c' \in C$ then intuitively the decision that yielded c is more definite than, or outranks, the decision that yielded c'.

We say a policy model N is a *priority refinement* of M if for all requests sequences σ, $s(N, \sigma) \geq s(M, \sigma)$. Note that priority refinement depends on D, C and g. If we assume a fixed D, C and g we write this as $N \leq_s M$. □

Example 5 *If we are using the prioritized logic voting mechanism (D_p, f_p) then we can set C to be the set of non-negative integers and $g(V)$ to be the maximal priority of all the votes in $V \subset D_p$.*

The definition of stream refinement is partly unsatisfactory because it relies on the internal details of the implementation since it is based on a function g which gives the importance of a particular decision. It also seems to be too fine a relation. Let M be a policy model that only accepts the first three transaction requests. Let N be a policy model that only accepts the first three transaction requests that are under \$50. Then $N \leq_s M$ but we can design M and N so that $N \not\leq_p M$.

3.5 Analysis

The formal definition of policy automata and their properties gives us the ability to formally check policy automata for those properties.

3.5.1 Detecting Conflicts

If a policy model has a finite number of states we can 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. (If our policy model has an infinite number of states then we can make the number of states finite by using abstraction.)

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 [56] gives an algorithm for finding the consequences of a defeasible theory in time that is polynomial with respect to the number of literals and defeasible logic rules, and [47] gives a linear time algorithm.

3.5.2 Redundancy

We may also want to check that a policy automaton P is redundant in a policy model $M = (\Pi, D, f)$. Recall that a policy automaton is redundant in a policy model if adding it to the model does not change which transactions are approved or rejected. Let $M' =$

$(\Pi \cup \{P\}, D, f)$ be the model M augmented with automaton P. For a given policy model state q' of M' let $d_{q',t}$ be the vote that P gives when processing transaction t in state q', and let $V_{q',t}$ be the set of votes supplied by the automata in Π. The policy automaton P is *redundant at* q' if

$$\forall t \in T, f(V_{q',t} \cup \{d_{q',t}\}) \quad = \quad f(V_{q',t}) \tag{3.4}$$

Claim 6 *P is redundant in M if and only if it is redundant at each reachable model state in M'.*

Proof: $\Leftarrow$. We prove the if direction by induction on the length of input sequences. Recall that states in a policy model with k policy automata are k-tuples of the states of the constituent policy automata. We use the notation $\pi_P(q')$ to denote the projection of the $k+1$-tuple q' state of M' to a k-tuple q state of M where we ignore the state of the P policy automaton.

Assuming (3.4) holds for all reachable states in M', our induction hypothesis with n being the index of induction is

$$\forall \tau \in T^n,\ q'_0 \xRightarrow{\tau \uparrow \alpha}_{M'} q'_n \text{ implies } \exists q_n.\ q_0 \xRightarrow{\tau \uparrow \alpha}_{M} q_n \ \wedge\ q_n = \pi_P(q'_n)$$

where q_0 is the initial state of M and q'_0 is the initial state of M'.

- **Case**: $n = 1$. The input sequence $\tau = t$ for some $t \in T$. The initial state q'_0 of M' is a $k+1$-tuple of the $k+1$ initial states of the policy automata in M'. By definition, we get the initial state of M by projecting out the initial state of P to get a k-tuple. The votes submitted by M's automata depend only on the automata's states and the transaction submitted. Since each of M's policy automata's state is the same in q_0 and q'_0, given t as input the automata in M will all submit the same votes V in M and M'. Let d be the vote submitted by P in this initial state. From (3.4) we know that $f(V) = f(V \cup \{d\})$ so both automata emit the same response. A policy automaton's update function δ depends only on the current state of the automata, the current transaction request, and the response emitted, so the next state after q_0 in M must be the same as the π_P projection of the next state in M'. So the induction hypothesis holds for $n = 1$.

- **Case**: $n > 1$. A similar argument applies in this case. Given an $n - 1$ length sequence τ, the induction hypothesis tells us that after reading τ, M will be in a state q_{n-1} that is a π_P projection of the state q'_{n-1} of M', and that both models have emitted the same sequence of responses. Let t be the next transaction request. The votes submitted by M's automata will be the same in both models, and (3.4) shows that P's vote does not affect the resolution function. Therefore, the update will proceed identically for M's automata in both models, and the next state of M will be a π_P projection of the next state of M'.

By induction, (3.4) therefore implies that for any finite length input sequence both policy models will emit the same response sequence.

$\Rightarrow$. To show the only if direction we assume that P is redundant but there is a reachable state of q' of M' and a transaction request $t \in T$ such that (3.4) does not hold. Since q' is reachable there must be some input sequence τ and response sequence α such that $q'_0 \overset{\tau \uparrow \alpha\,'}{\Longrightarrow}_M q'$. Assume without loss of generality that τ is the shortest such input sequence. Since no shorter input sequence leads to a state which violates (3.4), using the inductive argument used above for the if direction of this proof we can show that after reading τ the model M will be in a state q that is a π_P projection of q', and that both models will have emitted the same sequence of responses as output. Therefore the policy automata of M will submit the same votes V as they do in M'. Since P is redundant in M, both M and M' must emit the same output sequence on input $\tau; t$. In particular, they emit the same response on t in states q and q'. Therefore $f(V) = f(V \cup \{d\})$ where d is the vote contributed by P in M' in state q' with input t. However, this contradicts our choice of t and q', showing that no such t and q' exist. This proves this direction of the only if, and so we have proved the claim. $\square$

We can therefore check for redundancy by finding all reachable model states of the larger model M' and verifying that each state satisfies equation (3.4). As discussed above, evaluating f for all transactions can be done efficiently.

This technique for checking redundancy can be leveraged to validate a policy. Suppose a complex set of policy automata $P_1, \ldots, P_n$ is supposed to exclude a certain class of transaction request sequences. For example, we want our policy model to exclude reject a transactions after three consecutive b events. We can write a simple policy automaton P' that rejects that class, then check to see if it is redundant in $P_1, \ldots, P_n$. If P' is redundant then we know that $P_1, \ldots, P_n$ will reject that trace. If the class of sequences is large or infinite, as is the case in our example, then this technique will be faster than checking undesired sequences one by one. The automaton P' therefore functions as a partial specification of the desired policy model.

We can check for P being strongly redundant in M by checking a stronger condition on all reachable states of $M' = M \cup \{P\}$:

$$\forall t \in T, \forall\, V' \subseteq D,\ f(V_{q',t} \cup \{d_{q',t}\} \cup V') \quad = \quad f(V_{q',t} \cup V') \tag{3.5}$$

Verifying (3.5) is not as straightforward as verifying (3.4), which just required two evaluations of the resolution function. However, as discussed in Section 3.4.2, there are special cases where checking that a vote is redundant is simple. For example, if there is a vote $d \in V_{q',t}$ such that $d_{q',t} \preceq_r d$ (recall that a $\preceq_r$ orders votes by comparing the sets of the antecedents of the defeasible logic rules) then the vote $d_{q',t}$ can be ignored. If such a d can be found for all $t \in T$ and reachable q' then equation (3.5) will hold.

3.6 Expressiveness

In this section we give some sense of how expressive the policy automata formalism is in a formal sense. In Chapter 4 we examine expressiveness is a less formal sense.

3.6.1 Translating to Classical Automata

In this subsection we discuss the expressiveness of the policy automata framework by comparing it to a classical automata formalism.

Mealy machines [30] are a form of finite automata which give output instead of merely accepting or rejecting. A Mealy machine is a six-tuple $M = (Q, T, \Delta, \delta, \lambda, q_0)$ where Q is the finite set of states with $q_0 \in Q$ the initial state, T is the set of input events, Δ is the set of possible output events, $\delta : Q \times T \rightarrow Q$ describes how the machine updates state, and $\lambda : Q \times T \rightarrow \Delta$ is the function which determines what gets written as output. On input $a_1; a_2; ..; a_n$, if the machine goes through states $q_0, q_1, .., q_n$, the output of the machine will be $\lambda(q_0, a_1); \lambda(q_1, a_2); ..; \lambda(q_{n-1}, a_n)$.

If we set Δ to be $\{0, 1\}$ then a Mealy machine essentially becomes a finite state suppression automaton, except that it outputs its approval decision instead of copying or suppressing an input event. Given a Mealy machine with $\Delta = \{0, 1\}$, it is easy to transform it into a finite state suppression automaton by taking the output sequence of 0's and 1's and composing it with the input sequence using the $\otimes$ operator defined in Section 3.1.4.

A fixed conflict-free policy model $M_p = (\Pi, D, f)$ with a finite number of states (where 'fixed' means that no more policies will be added to the model) can be translated to a Mealy machine $M_m = (Q_m, T, \Delta_m, \delta_m, \lambda_m, q_{m0})$ in a straightforward manner. Let $\Pi = \{P_1, .., P_n\}$ be the policy automata making up the model. We set $\Delta = \{0, 1\}$. The state set Q_m is set to $Q_1 \times \cdots \times Q_n$ where Q_i is the state set of the corresponding policy automaton P_i. In other words, M_m has the same set of states as the policy model M_p. The initial state q_{m0} is the initial state of the policy model. We set δ and λ to match the transition of policy model: if $q \overset{a\uparrow x}{\Longrightarrow} q'$ in the policy model (where x is either yes or no) then $\delta_m(q, a) = q'$ and $\lambda_m(q, a) = 1$ or 0, for $x =$ yes or no, respectively. Note that determining that the policy model in state q given input a will emit x and transition to state q' can be precomputed if we fix the set of policy automata; every policy automaton has a finite amount of votes it may submit, so there are only a finite number of possible vote combinations, and therefore only a finite number of arguments that will be given to the resolution function f, which is deterministic. Informally, if we only need to consider a finite set of defeasible logic votes we can pre-compute a table mapping each possible vote combination to the corresponding output of the resolution. Using this table in combination with the policy automata's transition functions we determine how the Mealy machine should respond to a given transaction request in a given state.

Instead of translating to a single Mealy machine, we can preserve the modularity of the policy model by translating to a set of communicating Mealy machines. We can create a Mealy machine for each policy automaton where each Mealy machine's λ function specifies the policy automaton's vote on a particular transaction request, and these votes are then read by a manager Mealy machine, which in turn outputs the result of the resolution function f, which is in turn read by the Mealy machines corresponding to the policy automata so that they can update their state accordingly. This phased update proceeds as follows:

1. The Mealy machines corresponding to policy automata read the current input transaction request and output a vote.
2. The manager Mealy machine reads all the current votes and outputs a 0 or 1, indicating whether the transaction should be rejected or approved.
3. The Mealy machines corresponding to policy automata read the 0 or 1 and update their state accordingly.

The Mealy machines corresponding to policy automata are constructed in a straightforward way. The manager Mealy machine can be encoded as a finite state automaton because, since we have fixed the policy automata, there are only a finite number of possible votes. The manager machine only needs to look up the result of the resolution function using a table like the one described in the previous paragraph.

A similar translation will not work if we want our policy model to accept arbitrary policy automata that have been translated to compatible Mealy machines—even if each policy automaton has a finite number of states. Judging whether a $+\partial \mathsf{yes}$ can be inferred set of defeasible logic rules is at least as hard as the graph reachability problem (we can represent the graph using literals for vertices and strict rules for edges), which is a non-deterministic log space complete problem. Therefore a manager Mealy machine cannot resolve arbitrary votes using a finite number of states.

We *can* construct Mealy machines that resolve votes in the binary (D_2, f_2) and three valued logic (D_3, f_3) voting mechanisms. The election voting mechanism (D_e, f_e) and the prioritized voting mechanism (D_p, f_p) cannot be resolved using a Mealy machine because each mechanism requires a resolver to store arbitrarily large numbers.

3.6.2 What the Model Cannot Express

The policy model formalism is restricted in a number of important ways. A policy model can only reason about the information available in the transaction requests—if a transaction request fails to identify the merchant involved then a policy about merchants cannot be enforced by the policy model. This section discusses some other fundamental restrictions on what the model cannot express or enforce.

Limits on Enforceable Policies

As discussed in Section 3.1, if we consider security policies to be predicates over sets of transaction sequences then a run-time monitor like a policy model can enforce a strict subset of security policies. A run-time monitor can only examine a single trace at a time, and only the prefix of the trace that has already taken place. This makes certain policies impossible to enforce. In [60] Schneider identifies a class of policies called *properties*, for which validity solely depends on a single transaction sequence, and a subclass of properties called *safety properties*, which are properties which in which every prefix of a valid transaction sequence must also be valid. As discussed in Section 3.1.4, safety properties are the class of policies which can be enforced by suppression automata, which encompass policy models. This excludes potentially useful policies like the following, some of which were discussed earlier:

- **Anti-bribery policy**: Purchases should not depend on previous payments to the cardholder. For example, payments *from* a merchant (event a) should not necessarily precede payments *to* the same merchant (event b). A single trace of the form $..; a; ..; b; ..$ is not a violation of the policy since the payment from the merchant may be a coincidence. However, if all the purchase histories containing a b were of the form $..; a; ..; b; ..$ then the policy would be violated. A policy model tracking transactions could not enforce this policy because it would require monitoring more than one sequence of transactions[4].

- **Global spending limit policy**: Employee purchases should not exceed $10,000. A company may distribute programmable payment cards, each linked to the same bank account, to 10 employees. The company does not want the employees as a group to spend more than $10,000. As was the case with the anti-bribery limit policy, enforcing this policy requires observing more than one sequence of transactions, and therefore cannot be enforced by a single policy model watching one card.

- **Loan policy**: Any money borrowed must be paid back. For example, a cardholder is allowed to borrow money—event b—so long as that money is payed back—event p—eventually. This policy is a property since the validity of a sequence of transactions does not depend on the other possible sequences. However, it is not a safety policy as there are valid sequences where a prefix of the sequence is not valid. The sequence $b; a; a; p$ is valid since the borrowed money is payed back, but the prefix $b; a; a$ is not valid, since money has been borrowed without being returned. A policy model cannot know that a cardholder will eventually return the money, and therefore cannot distinguish between $p; a; a; b$ and $p; a; a$ before allowing the cardholder to borrow money[5].

[4]The anti-bribery policy is similar to an information flow policy about program control flow: we would like to ensure event a occurring does not necessarily cause event b.

[5]The loan policy corresponds to a liveness property for program behavior—we require that event p eventually leads to response b.

- **Alcohol purchase policy**: Alcohol can only be purchased with a meal. An employer may wish to restrict alcohol purchases. An alcohol purchase (event a) can only take place just before or after food has been purchased (event b). Consider a transaction request a for alcohol which is made before any food purchase. A policy model cannot authorize the purchase because the cardholder may never buy the required food. However, the cardholder may be planning to purchase the food immediately after the a transaction request takes place. As with the previous policy, the policy model cannot enforce policies in which one event can only take place if a certain future event also takes place.

If we restrict policy automata to only have a finite number of states, we cannot enforce any policies that require counting or storing unbounded information. For example, consider a trading on a margin account where a cardholder can open an account (event a), borrow a dollar (event b), do some investing (event i), pay back a dollar (event p) and close an account (event c). A policy requires that every dollar that is borrowed must be a paid back before the account is closed. A sequence of $a; b; b; i; p; p; c$ is permitted but $a; b; b; i; p; c$ is not. In general, the policy requires that a transaction sequence must contain at least as many p's as b's. Since this would require counting the number of b events this policy cannot be enforced by a finite state policy model. (However, if we put some bound on the number of dollars borrowed it is possible to enforce the policy.)

Storing and Retrieving Information

A natural feature one might want to add to a payment card is a system for logging important events. For example, it is in a merchant's interest to track a customer's purchase patterns—such information can yield efficiencies in future stocking and promotional strategies. A merchant could install an applet on a programmable purchase card (presumably with a customer's consent in exchange for a discount or other special treatment) that records the transaction events that occur with the card. At some point in the future the merchant will retrieve this event log from the card. The policy automata model does not cover such behavior. Chapter 4 discusses some ways a policy automata described in the Polaris language used by our tool can interact with arbitrary Java code using an *imported function*. However, our model does not account for any extra API that gives access to information about the applet's state. The only side-effects that our model considers are those that affect future accept/reject decisions.

Transaction Model

The policy model uses a simple transaction model where transaction requests are either approved or rejected and then never considered again. One can imagine richer models where a transaction request could be conditionally approved and then approved or rejected later based on new information. (Such a model would allow 'transactions' in the database sense of the word: a sequence of events that may not be simultaneous but are grouped together as an atomic operation.) For example, it would be useful if an on-card policy that recognizes the card is being used fraudulently could re-examine previous purchases and retroactively reject those that now appear to be fraudulent. On the other hand, we may wish to retroactively approve a purchase that could not have been authorized earlier; the alcohol policy of Section 3.6.2 could be enforced by conditionally approving an alcohol purchase and then confirming this purchase when it is clear the alcohol is being purchased as part of a meal.

Communication Between Policy Automata

The composition mechanism in the policy model formalism allows little communication between policy automata; a policy submits an anonymous vote, which may affect the result of the resolution function, and this result gets passed to other policy automata. This restricted communication channel was by design, as a simple composition mechanism makes analyzing the system easier, and many useful policies can still be represented as policy automata (as demonstrated in Chapter 4). However, we can imagine situations where a richer form of communication may be desirable.

For example, we could allow a policy automaton to exchange messages of some sort with other policy automata. This would allow policies to gather information about what other automata are present in the model, or what information has been gathered by other policies. This data could be used to determine which vote to submit. For example, consider a policy automaton $A_{\max}$ that wants to maximize the amount of money spent. Normally, such a policy would vote to approve all transaction requests. However, if $A_{\max}$ detected another policy automaton A_3 which limits the cardholder to at most three purchases, then $A_{\max}$ could modify its vote so that low cost purchases are rejected. This would encourage the cardholder to spend more on the allowed three purchases.

Similarly, a policy automaton could delegate certain information gathering responsibilities to another policy automaton. For example, several policy automata $A_1, .., A_k$ may want to adjust their votes and current states depending on whether or not the transaction requests is considered to be an emergency—for example, a payment in a hospital emergency room. However, the decision on what constitutes an emergency may require a complex examination of the past sequence of transaction requests. Reproducing such functionality in all k automata is inefficient—it would be simpler to have one automaton which decides whether a given event constitutes emergency and then broadcasts that information to the other automata. (In Section 4.2 there is an example that shows how this can be approximated using the defeasible logic voting mechanism, but the information can only be sent to the resolution function, not directly to the automata. A similar partial solution is possible using imported functions, also described in Section 4.2—but such functionality would be outside the formal model.)

Our communication mechanisms could be enhanced with some sort of authentication. In the policy model formalism any policy automata can submit any vote. It could be useful to restrict votes based on the automata that submit them. This could be used in conjunction with the emergency signaling policy automaton described above—only a privileged policy automata could submit a vote or broadcast a message indicating that a given transaction request is an emergency. This authentication mechanism could limit the disruption caused by a user adding a poorly designed or malicious policy automaton to the card.

3.7 Summary

In this chapter we presented the security automata formalisms of Schneider [60] and Ligatti et al. [41] and adapted and extended them for our programmable purchase card application. We showed that many of the policy classes for run-time monitors (for example, liveness) have corresponding policies in the world of purchasing policies. Our extensions included the notion of *graceful enforcement*, which requires an automaton to make minimal changes while ensuring that a cardholder obeys a policy. We proved that suppression automata are capable of gracefully enforcing all safety properties.

Ligatti et al. do not discuss composition of suppression automata. We showed how a naive composition is problematic. To solve this problem, we use a voting mechanism based on defeasible logic to compose individual *policy automata* into a *policy model* which is essentially a suppression automaton. This formal model is capable of describing an overall purchase policy as a composition of smaller modular sub-policies.

With this formal definition in hand we investigated various formal properties that correspond to real world properties that are of interest to policy designers: conflict-freedom, redundancy and refinement. We also used this formal definition to illustrate the limits of our policy enforcement mechanism.

Chapter 4

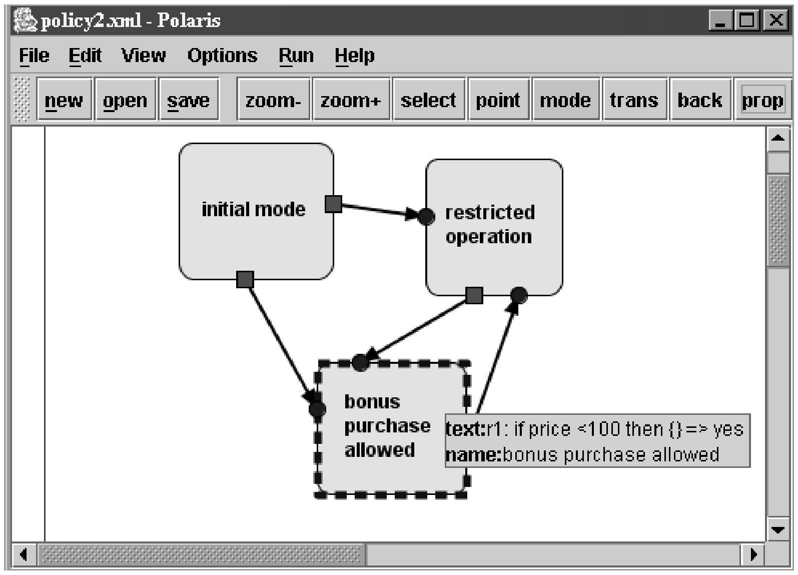
Language

Instead of encoding policy automata as a set of mathematical entities as described in Section 3.3.5 we use a less cumbersome graphical language that is closer to popular modeling languages. There is a straightforward correspondence between this language and the mathematical representation discussed in Chapter 3. In this chapter we present this language and discuss its suitability for encoding purchasing policies from a more empirical and engineering-centered perspective, in contrast to the more formal discussion in Chapter 3. We discuss a number of example policies, including a set of policies taken from a real enterprise purchase card. We also discuss our defeasible logic voting mechanism and compare it to other voting mechanisms.

4.1 Description of the Language

We split the state of a policy automaton into two components: *modes* and *variables*. If M is the set of modes and X is the set of possible values stored by variables then the automaton's set of states is $Q = M \times X$. Modes are akin to control points while variables record data.

The language is a mix of graphical and textual notation. Figure 4.1 shows the graphical interface used to create a policy model. We present the syntax of the language in this


policy2.xml - Polaris
File Edit View Options Run Help
new open save zoom- zoom+ select point mode trans back prop
initial mode
restricted operation
bonus purchase allowed

text:r1: if price <100 then {} => yes
name:bonus purchase allowed

Figure 4.1: Polaris automata editor

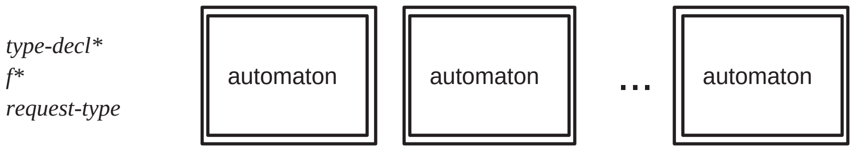


Figure 4.2: Structure of a policy model

section using a both graphical and text elements. The graphical elements are presented in figures while the textual elements are described in Table 4.1. We use the notation Z^* to mean zero or more Z's in sequence. We use the semicolon ";" as a separator for concatenated elements, and "|" indicates a choice between elements.

Figure 4.2 shows the syntax used to specify a policy model. A policy model consists of four elements: an optional list of type declarations, a optional list of imported function types, a type specifying the transaction request structure, and a set of policy automata. The type declarations, imported function types and transaction request type are specified textually–their syntax is given in Table 4.1. The type declarations allow a policy developer to define types that are useful for the policies; for example, a policy developer can define an enumerated type with three values {EU, US, OTHER} that indicate where a purchase is taking place. A developer can also define arrays and record types with fields to make

Type Declaration	*type-decl*	::=	`type` id `is` τ
Imported Function	f	::=	`import` $id : \tau \times \cdots \times \tau \rightarrow \tau$
Request Type	*request-type*	::=	`request is` τ
Type	τ	::=	$id \mid bool$
Enumerated Type			$\mid [id, .., id]$
Range Type			$\mid (n..n)$
Array Type			$\mid$ `channel`$[n, \tau]$
Record Type			$\mid$ `record`$[id : \tau; ..; id : \tau]$
Number	n	$\in$	1,2,...
Identifier	id	$\in$	the set of non-numeric strings
Variable Declarations	*var-decl-list*	::=	$id := e : \tau; ..; id := e : \tau$
Guard	*guard*	::=	e
Action List	*action-list*	::=	$a := e; ..; a := e$
Assignee	a	::=	$id \mid a.id \mid a[e]$
Vote Statement	*vote-stmt*	::=	`if` e `then` vt
Vote	vt	::=	$r; ..; r$
Defeasible Logic Rule	r	::=	$l, .., l \rhd l \mid \{\} \rhd l$
DL Implication	$\rhd$	::=	`->` $\mid$ $=>$ $\mid \sim>$
DL Literal	l	::=	$id \mid\ \sim id$
Expression	e	::=	`true` $\mid$ `false` $\mid n \mid id \mid e\ op\ e \mid\ -e$
			$\mid id(e, .., e) \mid e.id \mid e[e] \mid\ \sim e$
			$\mid$ `if` e `then` e `else` e `fi`
Operator	op	::=	$+ \mid - \mid \& \mid \vee \mid == \mid != \mid < \mid > \mid \leq \mid \geq$

Table 4.1: The textual elements of the language used to encode policy models

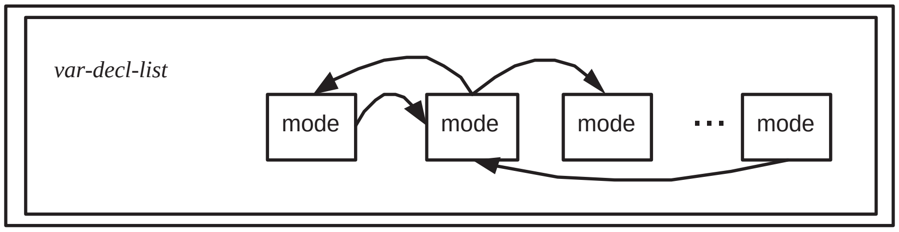


Figure 4.3: Structure of a policy automaton

storing data easier. For example, a developer could define a record type with fields for year, month and day so that dates can be recorded and modified conveniently. The imported function types specify which functions are available from the environment and what their arguments and return types are. Imported functions are discussed below in Section 4.1.1. The transaction request type specifies what information is available about the transaction– this determines what the set T of possible transaction requests contains. For example, a transaction request could be a record with three fields: the price of an item (for example, \$30), an identifier specifying the merchant, and an integer giving the current time. The transaction request is referenced using a special identifier "`t`". For example, in a policy automaton might set a variable with a statement "`x:=t.price`" which indicates that the variable `x` will be set to the price of the current transaction request.

As mentioned above, the policy automata are specified graphically by drawing a rectangle. Inside this rectangle the policy developer draws a set of rectangles representing modes, and arrows connecting the modes. The policy automaton rectangle can be annotated with some text indicating the variables stored by the automaton. Figure 4.3 shows the structure of an automaton. Both the type and the initial value is specified for all of an automaton's variables.

The δ transition function is specified by drawing arrows from one mode to another. Figure 4.4a shows the general structure of an arrow. Each arrow is annotated with a *guard*, which is a boolean expression involving the variables of the policy automaton, the transaction request and a special boolean variable "yes" which is true if and only if the last transaction request was approved. The boolean expression is similar to the expressions in

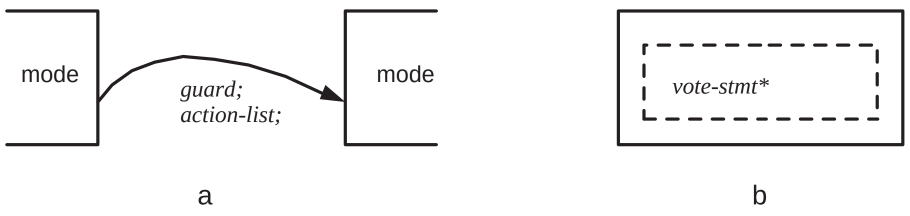


Figure 4.4: Structure of (a) an arrow and (b) a mode

high-level programming languages like Java or C. In addition to the guard, the arrow may contain a list of *actions,* which specify updated values for the variables. For example, an arrow from m to m' could be annotated with the guard "`t.price<30 & count==1 & yes`", where $count$ is a variable and t is a transaction request. It may also have a single element action list "`count:=2`". Such an arrow gives a partial description of δ, mapping (q, t, yes) to (q') where q is a state with mode m and the variable $count = 1$, t is a transaction request with a price under 30, and where q' is a state where the active mode is m' and the variables hold the same values as in state q except that $count$ is now 2.

There is a special arrow with no source mode that indicates which mode is the initial mode of an automaton.

The voting function γ is specified by annotating the mode rectangles with *vote statements*, as shown in Figure 4.4b. Each vote statement has a boolean expression (like the guard attached to arrows) referring to the current transaction request and the variables of the automaton, and a vote vt. If a policy automaton is in a mode m which is annotated with vote statement g and a transaction request arrives that, along with the current variable settings, makes the boolean expression true, then vote vt becomes the policy automaton's vote. Votes are lists of defeasible logic rules written in the syntax of the Deimos defeasible logic query tool [48]. Each vote statement therefore gives a partial description of γ. Figure 4.1 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 $\{\} \Rightarrow \mathsf{yes}$ written using ASCII characters. The rule essentially says "conclude yes tentatively unless others override."

4.1.1 Imported Functions

The Polaris language is intended to capture the core behavior of a policy which depends on the history of previous transactions. The language is not intended as a general purpose language for arbitrary control flow, data manipulation or logging. However, a policy may need to access or manipulate data in order to make decisions about permitting a transaction. We feel that such functionality should be implemented in a language appropriate for that class of behavior, and then integrated with policies described in the Polaris language.

The Polaris language offers an interface to general purpose programming languages through *imported functions*. An imported function is declared in the policy model and can be called in any expression in any policy automaton. Imported functions are intended to allow policy designers to incorporate functionality that cannot be expressed succinctly, or expressed at all, in the Polaris language. For example, an expression to check if a merchant is on a list of approved merchants can be written as "`isApproved(t.merch)`" instead of writing a long expression of the form

```
t.merch==SEARS ∨ t.merch==WALMART ∨ ...
```

A policy designer could also use an imported function to check cryptographic properties of transaction request data, something which would be difficult or impossible using Polaris' syntax.

The actual implementation of the function must be supplied through some mechanism external to Polaris—for example, it could be written by the policy developer—and compiled or linked with the Polaris-generated executable code implementing the policy automata. In our current implementation for the Java Card platform (described in detail in Chapter 5), the policy designer writes a Java Card compliant Java implementation of the function which matches the template generated by the Polaris compiler. This Java implementation is then combined with the Java files produced by the Polaris compiler before Java compilation.

4.1.2 Translation to Formal Policy Automata

The Polaris language is intended to be user-friendly way of specifying the formal model described in Chapter 3. Everything in the Polaris language can be easily mapped to the policy automata formalism from Section 3.3.5, with the significant exception of imported functions. The semantics of the language described in this chapter is defined by translating the language to the formal model, and then applying the semantics described in Section 3.3.6.

The set T of transaction requests is specified by the request type; if the transaction request has type τ then T is the set of possible values that a variable of type τ can take.

Recall that a policy automaton is a four-tuple (Q, q_0, γ, δ). A policy automaton in the Polaris language defines a set M of modes and a list of typed variables $v_1, \ldots, v_n$. Let X_i be the set of possible values the variable v_i may take. The formal automaton's set of states Q is simply $M \times X_1 \times \ldots \times X_n$. The initial state q_0 is $(m_0, v_{1,0}, \ldots, v_{n,0})$ where m_0 is the initial mode (as indicated by the special arrow with no source mode) and $v_{i,0}$ is the initial value of the variable v_i, which is specified when the variable is declared.

As mentioned above, the voting function $\gamma : Q \times T \to D$ is specified by the collected vote statements attached to each mode. Each mode's vote statement gives a partial description of γ, indicating how γ behaves for states composed of that mode. If no vote statement is attached to a mode m then γ maps all states where m is the mode component to a default empty vote containing no rules.

The transition function $\delta : Q \times T \times \{\textsf{yes}, \textsf{no}\} \to Q$ is similarly specified by the collected arrows in an automaton. Each arrow gives a partial description of δ, indicating how the function behaves in the source mode of that arrow. If an arrow starts at m and ends at m' with guard e and variable updates $v_1 := x'_1, \ldots, v_n := x'_n$ then $\delta(q, t, a) = (m', x'_1, \ldots, x'_n)$ if $q = (m, x_1, \ldots, x_n)$ and the arrow's guard evaluates to true when the variables, the transaction request t and the special yes variable (recall that yes is true if the request was approved) are substituted with their respective values. If no arrow starts from the mode of the current state, or if no such arrow has a guard that evaluates to true, then the state remains unchanged: $\delta(q, t, a) = q$.

Semantics and Analysis of Imported Function

If the policy model has imported functions then the γ and δ depend on what values the functions return at runtime. For example, if an arrow leading from m to m' has a guard “E(t)” where “E” is an imported function returning true or false, then (assuming the policy automaton has no variables) δ maps (m, t, a) to m' if “E” returns true.

Polaris makes certain assumptions about the behavior of imported functions. First of all, we assume imported functions will eventually terminate and return a value without throwing exceptions. We assume the function will return a value that is of the proper type. We also assume that the imported code will not interfere with the code generated by compiling the policy automata.

Even under those assumptions we cannot predict exactly how an imported function will behave. If we assume that the “E(t)” predicate satisfies our assumptions then it is clear how the policy automaton will behave at runtime, when the code implementing the predicate is available. However, our analysis algorithms cannot precisely model whether a given argument will satisfy the predicate.

There are a number of strategies for accommodating imported functions in the analysis. The analysis algorithm can leave such predicates uninterpreted and conservatively explores both possibilities. For example, the procedure that checks conflict-freedom would not actually check that a seller is on the list of approved vendors. Instead, the analysis checks that there are no conflicts whether or not the seller is approved. If this conservative analysis yields conflicts that are not actually possible then the policy writer can include simple constraints on the predicates to eliminate some spurious counterexamples, or bring some of the external code’s functionality into the policy automaton by replacing a call to `E(...)` with an expression of the form

```
    if (t.merch==SEARS) then true else E(t.merch) fi
```

so that Polaris can model enough of the function's behavior to avoid a spurious counterexample.

A policy developer may wish to mark an imported function as a true function (that is, the function will return the same value at different invocations with the same arguments), and Polaris could exploit this in the analysis or code-generation (for example, the results of such a function could be cached safely).

Some of the restrictions on the imported code, such as not modifying data that is used by the compiled automata or returning data of the proper type, is partially enforced by the Java type system—we mark variables in generated code as 'private', and the implementation of the imported code must match the method type or it will not compile. If the imported code throws checked exceptions then the Java compiler will show an error instead of compiling the policy (the imported code may still throw run-time exceptions). Other properties, such as termination, satisfying any constraints specified by the policy designer, or tighter constraints on what values the imported code may return, could perhaps be checked by a Java analysis tool like a model checker or static analysis tool. The Polaris code generator could generate JML[12] annotations in the templates of the imported methods to aid a policy designer validate an implementation.

4.2 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.

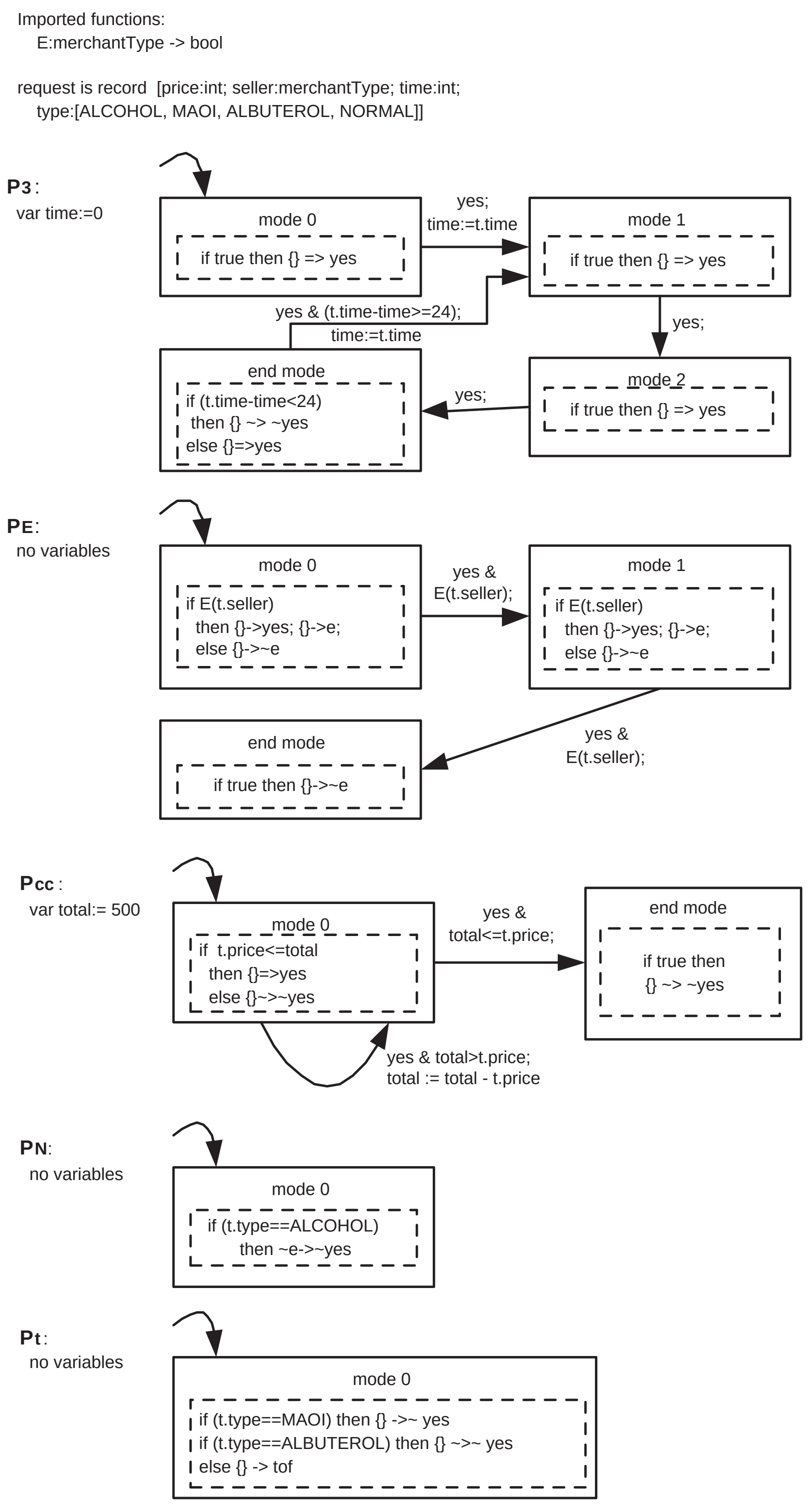


Figure 4.5: Example payment card policy model

The last policy, P_t, deserves some explanation. Tofranil is an prescription drug used to treat depression [62]. 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 cardholder 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 4.5 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). In this expression the "`E(..)`" is an invocation of an imported function that is supplied from a external library or implemented by the policy designer. 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* ; *action-list*". The *expression* indicates when that transition is enabled and the *action-list* 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 action-list 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 sketch how the policies in Figure 4.5 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 ~`yes`. Since ~`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

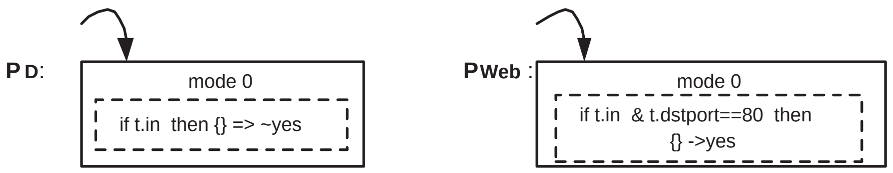


Figure 4.6: A simple firewall policy model allowing incoming packets destined for port 80

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.

4.3 Example: Network Access Policies

We think that our formal framework is general enough to be applied (perhaps with minor modifications) in domains other than payment cards. In particular, we feel that network access control is a suitable application. In this section we present some network access policies that have been encoded in the Polaris language.

A common firewall configuration blocks all incoming IP packets unless they are headed for a particular server that is listening on a specific port. For example, a firewall protecting a web server may block all packets that are destined for any port other than 80, the standard HTTP server port. We can represent such a policy using the two policy automata pictured in Figure 4.6. The automaton P_D sets the default policy: tentatively reject all incoming packets (that is, packets where the `in` field of the transaction request is set to

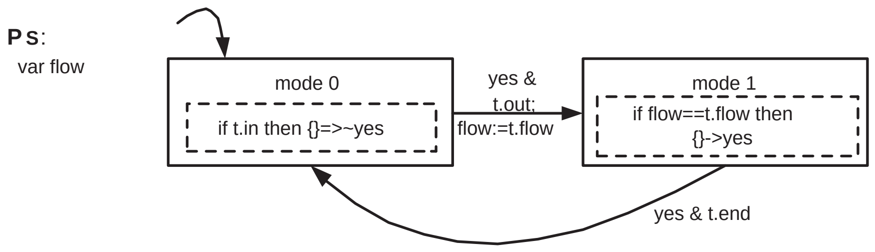


Figure 4.7: A stateful firewall policy automaton allowing incoming response traffic

true). The automaton P_{Web} overrides P_D (since its vote uses a strict rule) in the case that the incoming packet is destined for port 80.

The policies in Figure 4.6 do not have any non-trivial state. We can use a stateful policy automaton to model a policy that allows incoming packets only if they are responding to a previous packet that was sent out. Policies like this are used to allow users to contact external web servers without allowing external entities to contact internal servers. The policy is shown in Figure 4.7. This automaton tentatively rejects incoming packets until an outgoing packet is seen. The automaton then records the flow information for that outgoing packet and allows response traffic in that flow until a packet is seen that ends the connection, when the automaton returns to its original state.

We can use a similar automaton to specify a web server access policy. Consider a news web site that embeds images in its web pages. HTML allows a content developer to embed images from arbitrary URLs in a web page, so its relatively easy for other web sites to embed pictures from the news website in their own pages. The news web site may want to ensure (for licensing reasons, for example) that its images only appear embedded in its own pages. The policy automaton in Figure 4.8 implements such a policy. The automaton begins in a mode that refuses access for files with a JPG extension. When a client requests an HTML file the automaton moves to a state where all requests are tentatively allowed (allowing other policies to restrict access for other reasons).

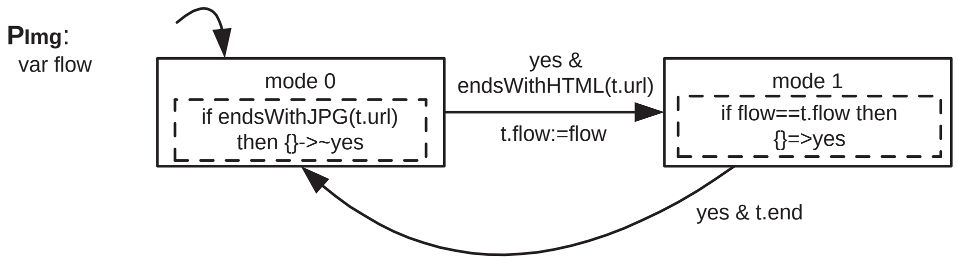


Figure 4.8: A web server access policy automaton protecting image files

4.4 Evaluation of the Language

In this section we attempt to evaluate the language using a number of approaches. In Section 3.6.2 we examined some of the formal limitations on what can be expressed using the policy model formalism. In this section we will use a more informal and empirical approach. We give some examples of broad classes of policies which can enforced with the language. We also examine the University of Pennsylvania's purchase card system and show how many of the policies for that system can be represented in the Polaris language. Finally, we illustrate the effectiveness of the defeasible logic voting mechanism by comparing it to some other voting mechanisms.

4.4.1 Policies That Can Be Encoded

The Polaris policy language is capable of representing several useful purchase policy types, including:

- **Approved/Rejected Merchants**: Polaris can encode a policy that only approves purchases from a list of merchants, or merchant types (for example, hotels or fast food restaurants). Polaris can also encode a policy that excludes purchases from a list of merchants or merchant types. Figure 4.9a shows an example of a policy that excludes purchases from airlines and taxicabs. The MCC field (Merchant Category

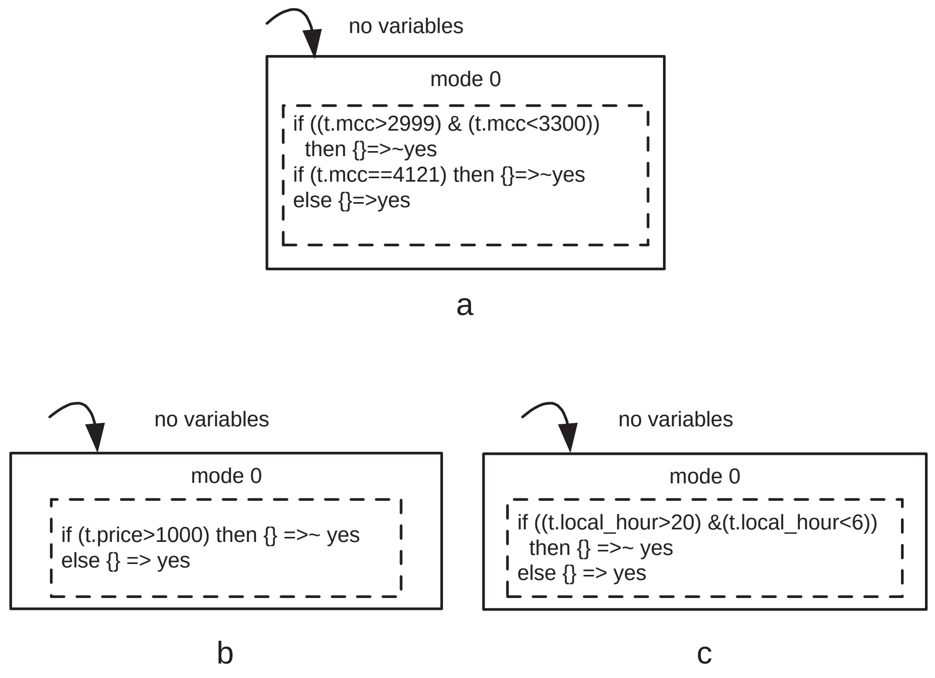

no variables
mode 0
if ((t.mcc>2999) & (t.mcc<3300))
then {}=>~yes
if (t.mcc==4121) then {}=>~yes
else {}=>yes
a
no variables
mode 0
if (t.price>1000) then {} =>~ yes
else {} => yes
b
no variables
mode 0
if ((t.local_hour>20) &(t.local_hour<6))
then {} =>~ yes
else {} => yes
c

Figure 4.9: Purchase policy automata for (a) rejecting certain classes of merchants, (b) imposing a per transaction spending limit, and (c) preventing purchases made at night.

Code) referred to in the policy is a standard identifier assigned by credit card companies to classify merchants by their type of business [33]. An MCC in the closed interval [3000,3299] indicates the merchant is an airline, and an MCC of 4121 indicates that the merchant is a taxi company.

- **Approved/Rejected Products**: We can encode a policy which excludes or includes certain products. The policy P_N in Section 4.2 is an example of such a policy.

- **Value Limit**: We can encode a policy that sets an upper limit for the money spent in a single transaction. Figure 4.9b shows an example of such a policy which ensures that no more than $1000 is spent in a single transaction. Additionally, we can encode a policy that resets the limit after a period of time—for example, allowing no more than $2000 to be spent in a week.

- **Limit the Number of Transactions**: We can encode a policy that limits how many transactions can take place using a card. As was the case for the value limit policy, we can reset the limit after a period of time. The policy P_3 in Section 4.2 is an example of such a policy, which has limits a card to three purchases per day.

- **Cash Card Policy**: Polaris can encode policies that limit a card to a set limit of all total purchases. The policy P_{cc} in Section 4.2 is an example of such a policy, which has a limit of $300.

- **Time Windows**: We can encode a policy which restricts purchases at certain times of day. Figure 4.9c shows a policy which prevents purchases between 9pm and 6am.

- **Drug Interactions**: Polaris can encode policies that guard against harmful drug interactions. Policy P_t in Section 4.2 is an example of such a policy concerning the drug Tofranil.

4.4.2 The Penn Purchasing Card System

In this section we describe the policies that the University of Pennsylvania (Penn) applies to use of its corporate purchasing card, the Procard. We then examine how these policies can be encoded as a Polaris programmable purchasing card.

In the twelve months to September 2004, Penn purchasing spent more than $82 million in more than 115,000 transactions, of which almost $464,000 was spent in more than 2200 transactions using the Procard [57]. Most Penn purchases are made through the BEN Buys and Penn Marketplace systems, which are electronic purchase systems in which trusted suppliers make arrangements to participate. The Procard is intended for purchases with vendors who have not yet taken steps to be included in the electronic marketplace. A separate purchase card called the Fleet Fuel Card is available for purchases of fuel and vehicle maintenance.

The policies governing the use of the Procard and Fuel Cards are described in the Purchasing Card Cardholder Guide [69] and on the Penn purchasing website (`http://www.purchasing.upenn.edu`). We have examined these documents and extracted a reasonably complete set of policies for the card. The Procard policies are summarized in Table 4.2 and the Fuel Card policies are summarized in Table 4.3. The "Commodities Matrix" mentioned in PC5 is a table showing which expenditures can be spent on which purchasing method. For example, alcoholic beverages must be purchased through Travel Office mechanisms (the Travel Office handles entertainment expenses) but cannot be bought through the BEN Buys system nor by using a Procard. Bottled water must be bought through the BEN Buys system, not the Procard. Books can be bought through either system, while box lunches can be bought only through the Procard. The Commodities Matrix lists 101 categories, of which 25 are permitted for Procard purchases.

At the time of writing, policy PC6 only applies to one supplier. Policy PC7 requires that purchases that can be made through Penn Marketplace must be made using that system instead of with a Procard. There are 102 suppliers available through the Penn Marketplace system. There are 213 suppliers on the list of deactivated suppliers mentioned in policy

PC1. Only the person listed on the card may use the card.

PC2. A suspended or terminated employee may not use the card.

PC3. Purchases must not be for the sole benefit of the cardholder.

PC4. Purchases must not be split into multiple transactions to avoid a per transaction threshold.

PC5. Purchases must only be made for commodities listed as permissible in the Commodities Matrix.

PC6. Purchases must be not made from a supplier who has refused to take part in the Penn Marketplace program.

PC7. Purchases must be not made from a supplier who is taking part in the Penn Marketplace program.

PC8. Purchases must not be made from suppliers who have been deactivated from the BEN Buys Supplier Database.

PC9. The total spending in a month must not exceed $5,000. If necessary, this limit can be raised with approval from the appropriate senior financial officer.

PC10. A single transaction must not exceed $1,000.

PC11. No more than 800 transactions can be carried out per month.

PC12. No more than 25 transactions can be carried out in a single day.

Table 4.2: University of Pennsylvania purchase card policy

PC8.

There are two kinds of Fuel Cards available. One is intended only for fuel purchases, while the other is for fuel and maintenance or repairs for Penn vehicles. In Table 4.3, policy F1 comes in two variations, only one of which applies to a given card; a card will either enforce F1a or F1b. Similarly, policy FM1 comes in two variations, one of which applies to a given card.

In Table 4.4 we show the degree to which each of these policies can be encoded in Polaris for use on a programmable payment card. Since policies FM1-7 are essentially the same as policies F1-7 we omit them from the table—any policy FMn can be encoded if and only if the policy Fn can be. We classify policies using two criteria: whether the policy can be exactly encoded or just approximated, and how much we would have to extend the payment infrastructure. For the first criteria, we mark the policy with an X if the policy can be encoded exactly—in other words, a policy automaton will allow behavior permitted by the policy and prevent behavior that violates the policy. We mark the policy with an A if the policy cannot be enforced exactly but we can approximate the policy; this approximation may allow some behavior that violates the policy and exclude behavior that the policy allows, but it can help discourage violations of the policy.

The second criteria indicates what kind of information must be supplied to the card for the card to enforce the policy. We assume that a programmable payment card infrastructure would make the following information available to the payment card, all of which is available in current credit card transaction records:

- The price
- The time and date
- An identifier specifying the merchant
- The appropriate Merchant Category Code (MCC)

We envision two possible extensions of this infrastructure:

Fuel Card policy for fuel-only cards:

F1. (a) The card can only be used by the person named on the card *or* (b) the card can only be used for one particular vehicle.

F2. The total spending in a month must not exceed \$2,500.

F3. A single transaction must not exceed \$50.

F4. No more than 50 transactions can be carried out per month.

F5. No more than 5 transactions can be carried out in a single day.

F6. The card can only be used for fuel purchases.

F7. Purchases must not be split into multiple transactions to avoid a per transaction threshold.

Fuel Card policy for fuel and maintenance cards:

FM1. (a) The card can only be used by the person named on the card *or* (b) the card can only be used for one particular vehicle.

FM2. The total spending in a month must not exceed \$5,000.

FM3. A single transaction must not exceed \$1,000.

FM4. No more than 50 transactions can be carried out per month.

FM5. No more than 5 transactions can be carried out in a single day.

FM6. The card can only be used for fuel, maintenance or repair purchases.

FM7. Purchases must not be split into multiple transactions to avoid a per transaction threshold.

Table 4.3: The policies of the University of Pennsylvania Fuel Card

	can be encoded?			
policy	*yes*	*w/ list of items*	*w/ identification*	*no*
PC1			X	
PC2				X
PC3				X
PC4	A			
PC5	A	X		
PC6	X			
PC7	X			
PC8	X			
PC9	X			
PC10	X			
PC11	X			
PC12	X			
F1a			X	
F1b			X	
F2	X			
F3	X			
F4	X			
F5	X			
F6	A	X		
F7	A			

X:	Policy can be encoded exactly.
A	Policy can be approximated.
w/ list of items:	Assumes card is supplied with data listing the items or services being purchased.
w/ identification:	Assumes card is supplied with authenticated information about the person or vehicle involved in the transaction.

Table 4.4: Which Penn purchasing card policies can and cannot be encoded as Polaris policy automata.

- **List of Items**: A list of items or services being purchased is available to the card.
- **Identification**: The card can authenticate the cardholder (perhaps through biometric means) and the vehicle being serviced (perhaps with an RFID or barcode on the vehicle).

We consider the first extension to be a minor extension, since for many businesses the list of items is readily available in electronic form.

Most policies can be enforced exactly with no modification to this infrastructure; these are marked with an 'X' in the 'yes' column. infrastructure. Other policies require one of the envisioned extensions. With a list of items available to a card, a policy automaton can exactly encode policies PC5, F6 and FM6 by checking that all the purchased items are on the list of approved commodities. Without information about individual purchases, a policy automaton can approximate the policy by assuming a correlation between the item being bought and the MCC of the merchant. For example, if the MCC of a transaction is a code for airlines then it is likely that a ticket for air travel is being bought—a policy automaton that forbids airline tickets can safely forbid any purchases from an merchant marked as an airline. The correlation between MCC and item is not always so strong. A gas station may sell snacks in addition to fuel, so a fuel card which allows charges from gas station MCCs may inadvertently allow some purchases which violate the gas-only policy F6. For stores like Walmart or Amazon, which offer a huge variety of merchandise, the MCC will give little information about what is actually being bought. We therefore mark policies PC5, F6 and FM6 with an 'A' in the 'yes' column because we can use the MCC-item correlation to enforce an approximation of the policy with no change in the payment infrastructure. Figure 4.10 shows policy PC5 encoded as a policy automaton. Since policies F6 and FM6 are so similar to PC5 we do not illustrate how they can be encoded.

Policies PC1, F1a, F1b, FM1a and FM1b are only possible to enforce if we have information about the identity of cardholders and vehicles being serviced.

Policy PC4 can only be approximated by a policy automaton. This policy is similar to the alcohol purchase policy (which forbids alcohol purchases made separate from a meal) discussed in Section 3.6.2—a violation of the policy only becomes apparent after multiple transactions have taken place, so a policy automaton has no reason to forbid the first transaction involved in the violation. However, we could encode a policy automaton that recognizes a series of transactions with the same merchant made in a short period of time that, when summed, breach the per-transaction threshold. For example, two purchases from the same merchant each for \$900 and made within 5 minutes of each other would probably be a violation of the policy since they seem to avoid the \$1000 transaction limit. Figure 4.10 shows an automaton that encodes this restriction. Such an automaton only approximates PC4, since a cardholder could wait a longer time between the transactions or buy the desired items from a number of merchants. We cannot imagine any reasonable extension of the payment infrastructure that would enable a policy automaton to better enforce PC4. Indeed, it is difficult to imagine any mechanical means for checking PC4 as it seems like there will always be cases where deciding if the policy has been violated is purely a matter of judgment.

Policies PC2 and PC3 are outside the scope of what can be enforced by a Polaris programmable payment card system. A smart card has no way of knowing whether an employee has been terminated, since the status of the employee can change without the change being communicated to the card. Nor can Polaris encode a policy that forces a cardholder to only purchase items that benefit Penn instead of the cardholder—it is difficult to imagine a smart card could infer in general that a purchase is being made that is not for Penn's benefit.

Of the 12 Procard policies, 10 can be encoded, either approximately or exactly, as Polaris policy automata in some form, possibly with additions to the payment infrastructure. In addition to the policies discussed above, Figures 4.10 and 4.11 shows automata for all the Procard policies that are encodable. Of the 16 Fuel Card policies, all 16 can be encoded as Polaris policy automata, though two policies can only be encoded as approximations of the original policy, and six of the policies require some additions to the

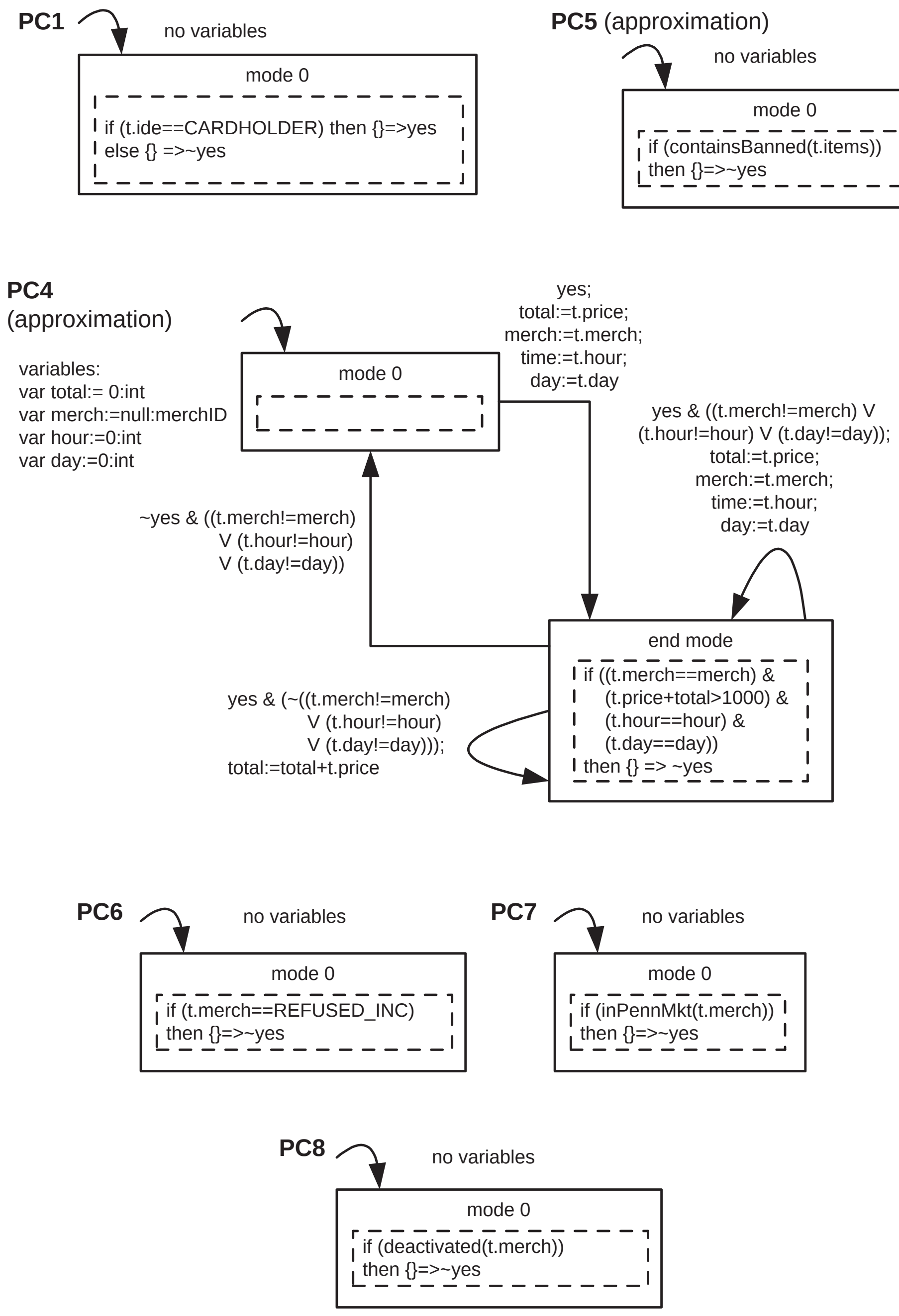


Figure 4.10: University of Pennsylvania Purchase Card policies encoded as policy automata (1 of 2)

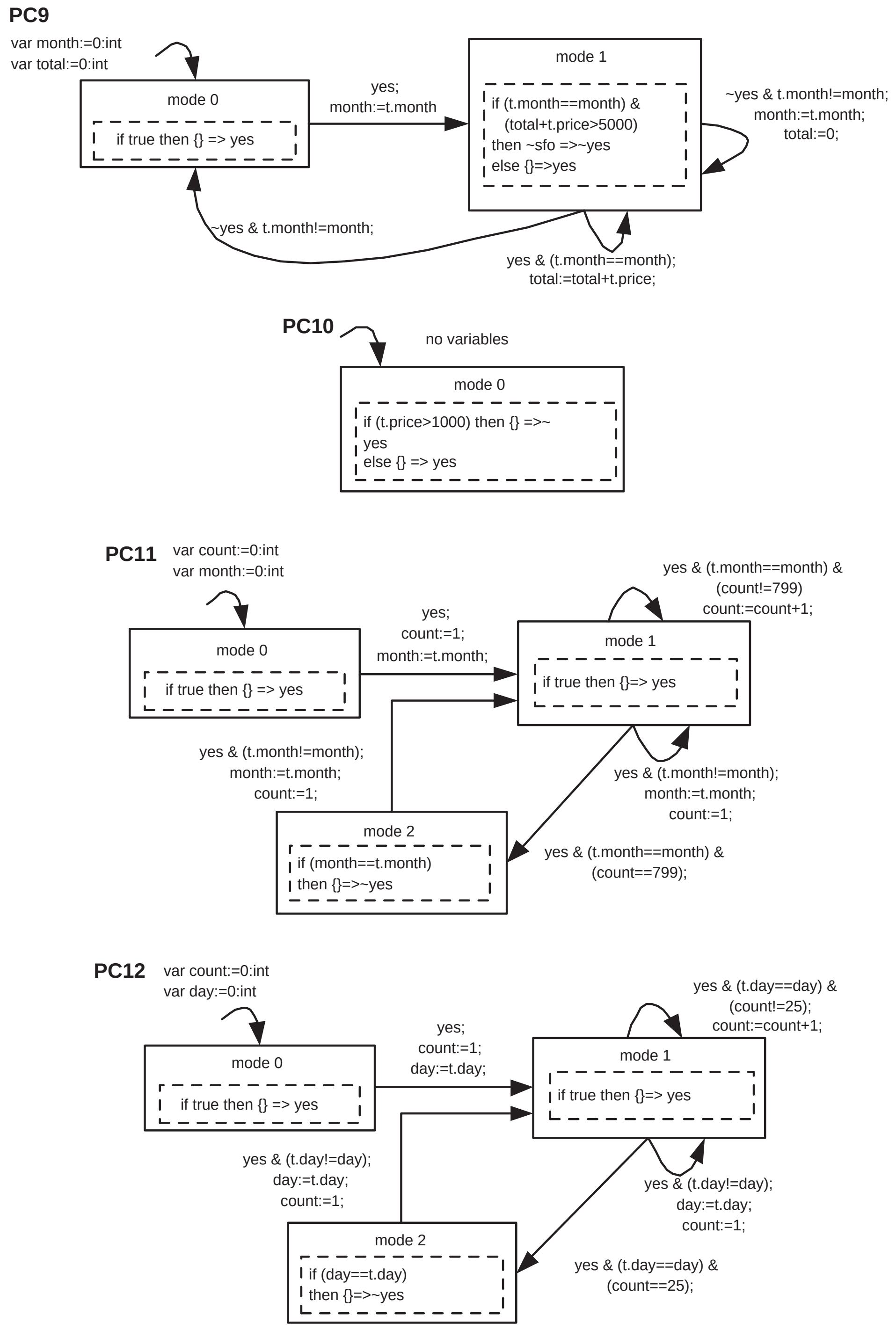


Figure 4.11: University of Pennsylvania Purchase Card policies encoded as policy automata (2 of 2)

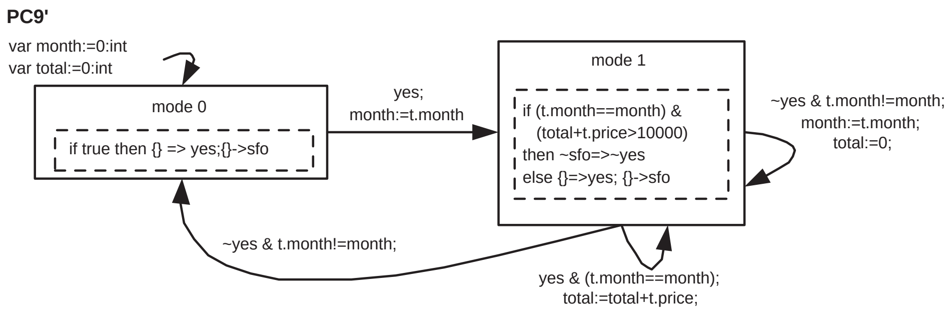


Figure 4.12: Modified PC9 automaton that overrides the initial PC9 automaton and allows a $10,000 per month spending limit.

payment infrastructure. We do not illustrate any of the Fuel Card policies as they are very similar to various Procard policies.

None of the policy automata discussed above use the ability of policy automata to override the votes of other automata. Most of the Penn Procard policies are written so that they cannot be overridden and they do not conflict with each other. The one exception is policy PC9, which states that the $5000 monthly spending limit could be raised if appropriate permission is given. The automaton for PC9 has been encoded with this overriding in mind: it makes its vote to reject purchases over its limit conditional on the absence of a literal sfo. Thus, PC9 will be overridden if we install an automaton such as PC9' shown in Figure 4.12. When this policy approves a purchase its vote asserts the literal sfo (indicating it was approved by a senior financial officer) and therefore blocks the vote of PC9. The card will therefore allow a cardholder to spend up to $10,000 a month.

4.4.3 Comparison and Discussion of Voting Mechanisms

In this section we evaluate the necessity and effectiveness of the defeasible logic voting mechanism by comparing it to the other voting mechanisms described in Section 3.3.4. The comparison is chiefly made by replacing the votes of the example from Section 4.2 with votes from each of the mechanisms, or by showing why such a replacement is impossible.

We are concerned in particular with the four policies $P_3, P_{cc}, P_E,$ and P_N. Recall that P_3 blocks future purchases after three purchases have taken place (for the discussion in this section we assume that all purchases are taking place within a single day so P_3 never resets its counter to allow further purchases). P_{cc} blocks purchases once 500 dollars have been spent. It is important to note that both P_3 and P_{cc} tentatively approve purchases which do not violate their respective polices. For example a card with only P_3 installed will approve the first three purchases and then deny further purchases. A card with both policies installed will allow any sequence of three purchases with a total cost bounded by \$500.

The policy P_E is an emergency policy that overrides P_3 and P_{cc} if an emergency is taking place. Thus, a card with P_E installed will approve any emergency purchase, even if the purchase violates the policies P_3 or P_{cc}.

Finally, P_N prevents purchases of alcohol (even if the purchase is one of the first three purchases or if does not exceed the \$500 total purchase limit). However, in the case of an emergency purchase, policy P_N defers to P_E; emergency purchases of alcohol are permitted.

In the following discussion we attempt to use alternate voting mechanisms to reproduce the behavior described in the preceding paragraphs.

Binary Voting Mechanism

Recall that the binary voting mechanism (D_2, f_2) has two possible votes, true and false, and votes are resolved by taking the conjunction of all the votes.

The binary voting mechanism cannot replace the defeasible logic voting mechanism in our example. Consider a card with P_3 and P_E installed, and two transaction request sequences $a; a; a; a$ and $a; a; a; e$ where a is an non-emergency purchase and e is an emergency purchase. Both sequences will potentially trigger P_3, which limits transactions to three per day. On the final a in $a; a; a; a$, P_3 must vote false in order to reject the fourth purchase. However, P_3 will submit the same vote for the e transaction in $a; a; a; e$, which will force the rejection of the transaction even though we would like P_E to override P_3; the binary voting mechanism gives us no way to override a policy.

3-Valued Logic Voting Mechanism

Recall that the 3-valued logic voting mechanism (D_3, f_3) allows three possible votes: true, false and $\perp$ (a sort of 'do not care' vote), and a true vote conflicts with a false vote.

Like the binary voting mechanism, the 3-valued logic voting mechanism fails to model our example. Again, consider a card with P_3 and P_E installed, and two input sequences $a; a; a; a$ and $a; a; a; e$. If the fourth a in the first sequence violates the P_3 policy then P_3 must supply a false vote to force a rejection of a. However, from P_3's perspective the same holds for the event e in the second, so the automaton will supply a false vote. This makes it impossible for the P_E automaton to force the acceptance of the e transaction—voting true will cause a conflict, and voting $\perp$ will let the transaction be rejected.

Election Voting Mechanism

Recall that the election voting mechanism (D_e, f_e) allows three possible votes: true, false and $\perp$, and the transaction request is approved, or disapproved, depending whether there were more true, or false, votes respectively.

We cannot simply replace the defeasible logic votes of the example with election votes. Consider an emergency alcohol purchase that is made made after three purchases and breaches the \$500 spending limit. Policies P_3, P_{cc} and P_N all disapprove of the purchase and their three false votes will counter the one true vote submitted by P_E. The emergency transaction will then be rejected, contrary to the desired outcome.

However, we can approximate the defeasible logic mechanism's capacity for overriding policies by submitting multiple votes. This can be done by installing k copies of a policy automaton. Since each of these automata will behave exactly the same as the other copies, this is equivalent to allowing a single automaton submit multiple votes—effectively assigning a numbered priority to each automaton. Using this technique, we can approximate our four policy example by using the following votes:

Policy	Approval Vote	Rejection Vote
P_3	$\mathsf{true} \times 1$	$\mathsf{false} \times 1$
P_{cc}	$\perp \times 2$	$\mathsf{false} \times 2$
P_N	$\perp \times 4$	$\mathsf{false} \times 4$
P_E	$\mathsf{true} \times 8$	$\perp \times 8$

In the table above, 'approval vote' is the vote an automaton submits when a transaction request satisfies the policy (for example, for P_3 the purchase is the first, second or third purchase) while the 'rejection vote' is the vote an automaton submits when a transaction request violates a policy. Since P_N only rejects bad purchases it supplies a 'do not care' vote of $\perp$ when a purchase is not an alcohol purchase—it does not actively approve non-alcohol purchases, leaving that to the other policies on the card. Similarly, P_E votes $\perp$ when a purchase is not an emergency, leaving the decision to approve or disapprove entirely up to the other policies on the card.

The votes listed above ensure that P_N overrides approvals from P_3 and P_{cc}. It also ensures that non-emergency transactions will only be approved if P_3, P_{cc} and P_N approve. The emergency policy, with its eight votes, can override the rejection votes from the three other policies listed.

However, in this scheme a policy model with a single P_{cc} will never approve a transaction request, while a similar policy model using a defeasible logic voting mechanism would approve and reject transactions as desired. This is in fact unavoidable—any choice of votes for policies P_3 and P_{cc} will yield either a policy that votes $\perp$ when it should approve a transaction, or the votes will not yield a rejection in the case where only one of P_3 and P_{cc} approve a non-emergency purchase. While this voting mechanism yields some of the flexibility of the defeasible logic system, it is unsatisfactory because of this problem, as well as the difficulty involved in choosing the votes. The choice of votes for P_3 and P_{cc} involved a fairly tedious case analysis of the possible votes, and the two policies could not be designed as isolated modules. The subsequent assigning of magnitudes to the P_N and P_E automata depended strongly on the votes chosen for the other two policies. If we added more low-priority policies we would have to replace P_N and P_E with higher magnitude policies (or add more copies of those policies).

Finally, it is difficult to imagine using this voting mechanism to encode the signaling capability that P_E and P_N use. The policy P_E asserts that the literal e is true when there is an emergency, and other automata can use that literal in their vote as to trigger conclusions that would not normally be triggered, or suppress conclusions. In this case P_N uses the e literal to suppress its vote to reject. This kind of simple signaling could not be conveyed using copies of true or false votes.

Prioritized Logic Voting Mechanism

Recall that the prioritized logic voting mechanism (D_p, f_p) is essentially the 3-valued logic mechanism extended by attaching priorities to the true and false votes. (This differs from the strategy of duplicating automata employed for the election voting mechanism because priorities are assigned to individual votes instead of policy automata.) The same automaton may submit a vote with priority 3 in one case and submit a vote with priority 7 in another case.

The prioritized logic voting mechanism is the most flexible of the alternative mechanisms considered as it is not difficult to find votes that yield the desired behavior:

Policy	Approval Vote	Rejection Vote
P_3	$\mathsf{true}, 1$	$\mathsf{false}, 2$
P_{cc}	$\mathsf{true}, 1$	$\mathsf{false}, 2$
P_N	$\perp$	$\mathsf{false}, 2$
P_E	$\mathsf{true}, 3$	$\perp$

Unlike the case with the election voting mechanism, this voting mechanism gives the proper behavior when only the P_{cc} policy is installed–purchases are approved until the total cost exceeds \$500. The votes listed in the table above reproduce exactly the behavior we expect from the four policies used in our running example, even when only one or two policies are installed. The election voting mechanism also required careful analysis to assign the priorities to the votes while this mechanism did not require much analysis. For the purposes of encoding the four policies $P_3, P_{cc}, P_N,$ and P_E the prioritized logic mechanism is as expressive as our defeasible logic mechanism. It could be argued that for these policies it is better than the defeasible logic voting mechanism, since it is easier to understand which vote will override other votes when priority is decided by a simple number instead of the complex defeasible logic inference algorithm.

We see two disadvantages of the prioritized logic mechanism. One is that, while prioritized logic can encode the four policies $P_3, P_{cc}, P_N,$ and P_E, it cannot encode the signaling behavior that defeasible logic allows. Consider the following policies:

- P_D: A policy that determines if the purchase involves prescription drugs and, if it does, blocks drug purchases except in emergencies. However, if the cardholder is allergic to a drug, purchases of that drug will always be rejected.

- P_e: A policy that determines if the purchase involves an emergency and, if it does, permits all purchases.

- P_{ap}: A policy which is installed if the cardholder is allergic to penicillin.

We assume that each policy is developed using special knowledge (perhaps proprietary) that enables them to learn certain information about purchases. P_D can determine if a transaction request involves a certain drug (perhaps by consulting a licensed list of drug product codes), but it is not capable of determining if a transaction request is an emergency. Similarly, P_e is capable of determining if a transaction request is an emergency, but cannot determine if a purchase contains prescription drugs. P_{ap} is not capable of discovering any information about the transaction request—it is only there to indicate some information about the cardholder.

We are concerned with the behavior of the card when presented with a penicillin purchase. If the cardholder is not allergic to penicillin (and therefore P_{ap} is not installed) then P_D should approve the purchase only in an emergency–essentially, P_D defers to P_e in emergencies. However, if the cardholder is allergic to penicillin, P_D should *override* P_e

instead of deferring, so that the penicillin purchase is blocked even in emergencies. The key problem is that the priority of P_D depends on the presence of P_{ap} on the card: with P_{ap}, P_D must have a higher priority than P_e; without P_{ap}, P_D must have a lower priority than P_e. Furthermore, P_D cannot determine before voting whether P_{ap} is present or if the transaction is an emergency, so P_D must submit a vote that works for each possibility.

We can assign defeasible logic votes that yield the desired behavior as follows:

- P_D: If the purchase is a penicillin purchase then vote "$\neg e \Rightarrow \neg yes;\ ap \Rightarrow \neg yes$"
- P_e: If the purchase is an emergency purchase then vote "$\{\} \Rightarrow yes; \{\} \Rightarrow e$"
- P_{ap}: Always vote "$\{\} \rightarrow ap$".

The votes use two atomic formulas e and ap as signals; they are intended to mean 'emergency transaction' and 'allergic to penicillin' respectively. The vote of P_D will reject a transaction if either e is not true or ap is true. The vote of P_e tentatively approves the purchase, but it also asserts that e is true, a signal to other automata (like P_D) that an emergency transaction is taking place. Finally, the policy P_{ap} does not say anything about approving or rejecting the purchase; it simply asserts that the ap signal is true.

We now examine the behavior of these policies when a cardholder attempts to buy penicillin in an emergency. There are two cases to consider: a card with P_{ap} installed, and a card that does not have P_{ap} installed. In the first case, P_{ap} asserts that ap is true, triggering P_D's rule $ap \Rightarrow \neg yes$, which blocks P_e's vote to approve the purchase. The purchase will be rejected as it should be, since the cardholder is allergic to penicillin. In the second case, the cardholder is not allergic to penicillin, so the P_{ap} policy is not installed on the card, and the vote "$\{\} \rightarrow ap$" is not submitted. In this case, neither of the two rules in P_D's vote will be enabled because neither ap nor $\neg e$ is true. So P_D's vote will not block the vote of P_e, which approves the transaction request. Therefore the penicillin purchase will be permitted, as it should be.

It is impossible to assign votes that yield the same behavior using the prioritized logic voting mechanism. The priorities of P_e's and P_D's votes cannot vary depending whether

P_{ap} is installed on the card. Therefore, when an emergency purchase of penicillin is made, either P_D or P_e will always override the other policy; if P_e's vote to approve an emergency purchase overrides P_D's vote to reject a penicillin purchase, then this will happen even if P_{ap} is installed, which is contrary to the desired behavior. In order to get the correct behavior we could add some way for policies to query the card to see what other policies are installed, but this would unnecessarily complicate the formal model. We could instead extend P_{ap}, P_e or P_D with extra functionality so that, for example, P_{ap} is able to determine if the purchase is a penicillin purchase and therefore vote to override all other policies. However, this would force us to duplicate functionality in different policies, leading to a less modular design, and possibly violating licensing or security requirements that limit which policies can learn about transaction requests—perhaps the reason only P_D can determine whether a purchase is a penicillin purchase is because it relies on an expensive proprietary algorithm which cannot be copied to multiple policy automata.

This penicillin example shows how our defeasible logic voting mechanism, in addition to having great flexibility in resolving conflicting priorities, allows policy designers to use sophisticated signaling and make votes conditional on such signals. The prioritized logic voting mechanism cannot be used to encode the same behavior without extending our formal model or making policies less modular by duplicating functionality.

A second disadvantage of the prioritized logic voting mechanism is that, historically, explicit priorities have been difficult to use in a distributed setting. Lupu and Sloman [42] write that

> meaningful priorities are notoriously difficult for users to assign and may result in arbitrary priorities which do not really relate to the importance of the policies. Inconsistent priorities could easily arise in a distributed system with several people responsible for specifying policies and assigning priorities.

Our defeasible logic mechanism attempts to minimize such problems by using a more declarative notation, which implicitly assigns priorities to votes. We feel that a vote which states "I tentatively approve" or "I definitely reject" is more natural and less arbitrary than “I vote yes with priority 6”, whose meaning depends on what priorities all the other policies have chosen. We believe that using defeasible logic rules, where the inference algorithm resolves many potential conflicts dynamically, could potentially yield a simpler and more modular system for policy design—though a fair comparison would require long term use of the defeasible logic voting mechanism to see if different policy designers collaborating in a distributed fashion would avoid the problems described by Lupu and Sloman.

Replacing the Voting Mechanism

In this section we have showed how the defeasible logic voting mechanism allows votes which can express various levels of priority, and can exhibit signaling that lets votes react to the presence or absence of other policies. None of the alternate voting mechanisms considered can express the full range of behavior that the defeasible logic mechanism can. However, if the full power of the defeasible logic voting mechanism is not needed, it may be desirable to use one of the simpler mechanisms which are probably easier to understand; many people can understand numbered priorities while few are familiar with defeasible logic. In some sense, the voting mechanism is a parameter in the formal framework and implementation presented in this dissertation; if we replaced the defeasible logic mechanism with another mechanism much of the discussion of the formal framework would be largely unchanged—for example, the algorithms for checking conflict-freedom and redundancy would work for any mechanism. Similarly, the syntax of Section 4.1 and the implementation presented in Chapter 5 could be easily modified to use a different voting mechanism.

4.4.4 Functionality Outside the Scope of the Language

There are several capabilities that one might want to include in a programmable payment card which are left out of the language for the sake of simplicity. Data processing is one such capability. Polaris is not intended to model complex data processing functionality—the language is focused on control flow instead of manipulating data. However, there are cases where such functionality would be useful in a policy. One example is querying data structures—many of the policy automata presented in this chapter query a list of acceptable or forbidden suppliers or items. Another example of data processing that one might want in a payment card is some sort of logging functionality, where an on-card applet records consumer preferences and reports that data at some point in future. Finally, as mentioned in Section 4.1, a security policy may use cryptographic operations to check that a transaction request is permissible. The Polaris language is not intended to model such functionality.

From an engineering standpoint, however, we recognize the need to allow such functionality in policies that are encoded in the Polaris language. This motivated our inclusion of the ability to call "imported" code, as described in Section 4.1. The behavior of this code is not considered by the model in any detail—we assume that these method calls terminate and do not modify the state variables used to represent the automaton state, but any other behavior is not modeled.

4.5 Summary

In this chapter we presented a language which can be used to easily encode policy enforcers in a form that corresponds to the formal definition of policy automata discussed in Chapter 3. This language bridges the gap between a practical tool for a policy designer and our formal model.

We justified the language, and therefore the underlying formalism, by showing how it can encode a range of desirable policies, including 10 out of 12 policies that regulate the use of Penn's purchasing card, and 16 of 16 policies that regulate the use of the university's fuel cards. Additionally, we showed example encodings of many other policy classes, from cash card policies to policies that protect against harmful drug interactions. We justified our defeasible logic voting mechanism by showing how other mechanisms fail to concisely represent a given suite of policies.

Chapter 5

Implementation

We have implemented a prototype of the *Polaris* system 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 5.1. The tool is implemented in Java and is partly built using the Hermes [3] code base. The tool is almost 38,000 lines of code, not including the graphics library used for editing the automata.

5.1 Architecture

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 4.1 shows a screen shot of the automata editor.

Analysis engine: The analysis engine takes a policy model from the front end and

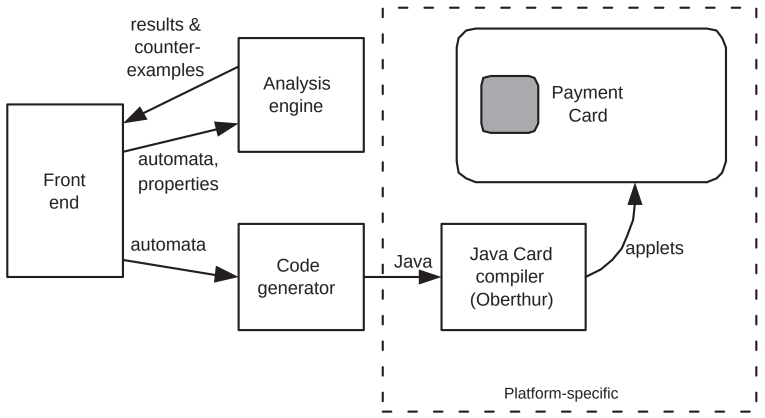


Figure 5.1: Polaris architecture

checks that the automata satisfy various properties the designer chooses. Currently we have implemented a conservative procedure for checking conflict-freedom.

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 transaction via a PC that has a smartcard reader. Before the transaction takes place the policy model implementation must approve the purchase request.

We use simple typing rules to check if expressions involving policy automaton variables and the transaction requests are well typed. 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 Analysis

In our implementation we have implemented a conservative version of the conflict checking algorithm of Section 3.5. In the Polaris language all of a policy automaton's votes are explicitly listed in the automaton's vote statements. Our algorithm gathers the votes of each automaton in a model and checks all combinations where one vote is picked from each automaton. If no combination leads to a conflict then we can be sure that the policy model is conflict-free. However, false positives are possible; if we find a combination that does lead to a conflict it may the case that no reachable state leads to that combination of votes. In most of the complex policy models we have examined the number of states is much higher than the number of combinations of votes, so this algorithm can be a cheap way to check for conflicts.

5.3 Code Generation and the Java Card Platform

The Java Card platform allows multiple applets to be installed on a single Java Card. Applets can communicate with each other using procedure calls and shared objects. Applets are protected from each other with a firewall mechanism, whereby inter-applet procedure calls are mediated by the Java Card system and shared objects must be explicitly labeled as shareable. This architecture yields a natural correspondence with our policy models: each policy automaton can be compiled into an applet so that policy automata can be added or removed dynamically.

There are two types of Java Card applets that need be generated: the *manager* applet and the *policy* applet. Figure 5.2 gives an overview of the code generation process. 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. Most of the manager applet's code deals with Java Card and transaction protocols; this

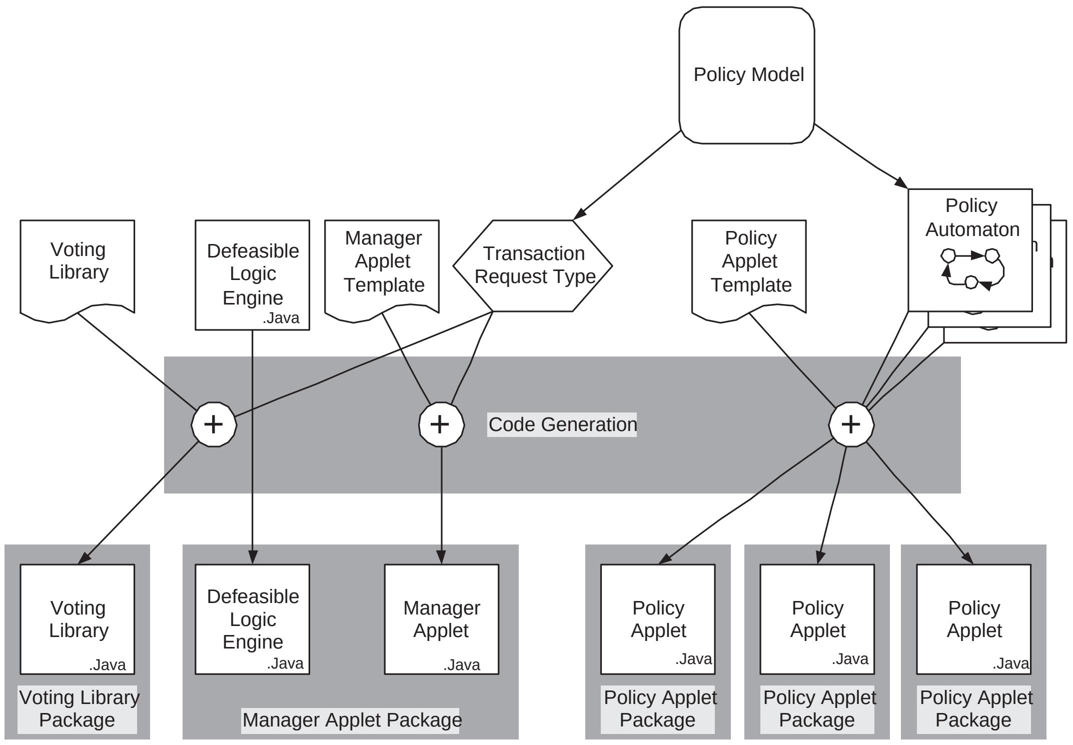


Figure 5.2: Polaris code generation process

code is specified as a template that is constant for all manager applets. We envision different applications using different transaction request types (for example, in a prescription drug payment system the transaction data may include information about which drug is being bought, while a credit card purchase system may only include information about the merchant, price and date) so we automatically generate the manager applet code that processes the transaction request data, adjusting it to the specific transaction request type. Similarly, we adjust the voting library which is shared by the policy applets and manager applet so that it can handle the appropriate transaction type.

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. Many cards require that all the memory an applet will need be allocated when the applet is installed. 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. In the code generation process we convert votes into a binary format that can be stored and read efficiently.

Figures 5.3 and 5.4 show the P_E emergency policy from Figure 4.5 after the code generator has translated it into Java code. We have omitted some of the code that deals with the Java Card platform as this is common to all applets and would make the figure much larger.

```
public class PolicyApp0 extends Applet
                      implements ApprovalInterface {
  private static byte pls_mode_var;
  private static VoteImpl vote0, vote1;
  private static byte[] t;
  public Vote getVote(byte inByte0, byte inByte1) {
      t[0] = inByte0; t[1] = inByte1;
      switch (pls_mode_var) {
      case 0:
          if ((PlsImported.E(t[0]))) { return vote0; }
          if (true) { return vote1; }
          break;
      case 1:
          if ((PlsImported.E(t[0]))) { return vote0; }
          if (true) { return vote1; }
          break;
      case 2:
          if (true) { return vote1; }
          break;
      default:}
      return NullVote.constructNullVote();
  }
  public static void install(byte[] bArray,
                       short bOffset, byte bLength) {
      (new PolicyApp0()).register(bArray,
          (short) (bOffset + 1), bArray[bOffset]);
      vote0 = new VoteImpl((byte) 0,
                Vote.STRICT, new byte[] {},
                (byte) 101,Vote.STRICT,new byte[]{});
      vote1 = new VoteImpl((byte) -27,
                      Vote.STRICT, new byte[]{});
  }
```

Figure 5.3: Java code generated from the emergency policy P_E (1 of 2).

```
  public void update(byte inByte0,
                     byte inByte1, boolean yes) {
      t[0] = inByte0; t[1] = inByte1;
      switch (pls_mode_var) {
      case 0:
          if ((yes && (PlsImported.E(t[0])))) {
     pls_mode_var = 1; }
          break;
      case 1:
          if ((yes && (PlsImported.E(t[0])))) {
             pls_mode_var = 2; }
          break;
      case 2: break;
      default:
          ISOException.throwIt(ERR_UNKNOWN_MODE);
      }
  }
  public boolean select() {
    AID Transaction_aid1
       = JCSystem.lookupAID(TRANSACTIONAPP_AID,
            (short) 0, (byte) TRANSACTIONAPP_AID.length);
    if (Transaction_aid1 == null) {
        // Cannot find the TransactionApp AID
        ISOException.throwIt((short) 0x0010);
    }
    TransactionInterface sio1 = (TransactionInterface)
        (JCSystem
         .getAppletShareableInterfaceObject(
            Transaction_aid1, (byte) 0x00));
    if (sio1 == null) {
        ISOException.throwIt((short) 0x0020); }
    // Register the policy to TransactionApp
    sio1.addPolicyApp();
    return true;
  }
}
```

Figure 5.4: Java code generated from the emergency policy P_E (2 of 2).

The policy and manager applets use a shared library of classes that contains the data structures and functionality needed to encode votes and allow the manager applet to query the policy applets' votes. More examples of the output of this code generation are available at the *Polaris* web site (`www.cis.upen.edu/˜mmcdouga/polaris`).

5.4 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 automata and check that the new combined policy model is conflict-free (or whatever property is desired). If the desired properties hold, the developer follows the steps described in [22], 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.

This illustrates the trade-off involved in adding policies dynamically versus installing them as a group; if policies are installed as a group it is easier to verify that they work correctly and do not run into conflict states. If policies are added one-by-one the user must re-check every policy addition to see if it introduces new conflicts.

Policies outside the card

Keeping policies on the card makes it easy for the user to manage their purchasing restrictions. However, there may be situations where the policies need to be kept somewhere else where they can be consulted for every purchase—for example, at the bank's transaction processing server, or at a special server operated by the enterprise or family that owns the card.

Such an architecture would still benefit from a standard formal policy framework. A bank may want to examine policies that are sent by users to ensure they do not damage the transaction approval functionality; this task would be less costly if the analysis could be done automatically using a Polaris-like tool.

	memory required on card (bytes)
original SET	11291
modified SET	15586
increase due to modification	25%

Table 5.1: Code size for original and modified SET manager applet

5.5 Experimental Results

5.5.1 Applet Size

A smartcard's limited memory makes code size an important consideration. To measure the impact of our policy integration scheme we augmented Lyubich's Java Card implementation of the of the SET protocol and measured the code size before and after the augmentation. Table 5.1 shows how much the applet size increased 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 how much EEPROM memory the applet occupied on a Oberthur GalactIC Java Card[1]. After extending the SET applet with a defeasible logic engine and the code necessary to manage policy applets the total applet size is only 25% larger.

A policy applet takes up additional space on a Java Card when it is installed. A feasible programmable purchase card architecture must have applets which are small enough to put a number of policies on a smartcard. Table 5.2 shows the size of the five applets generated from the automata in Figure 4.5. Once we have loaded the Java Card system software, the voting library and the manager applet, the Oberthur GalactIC card has room for about 33 policies with a mean size of 678.4 bytes.

Policy	memory required on card (bytes)
P_3	704
P_E	704
P_{cc}	672
P_N	608
P_t	704
mean	678.4

Table 5.2: Code size for selected policy applets

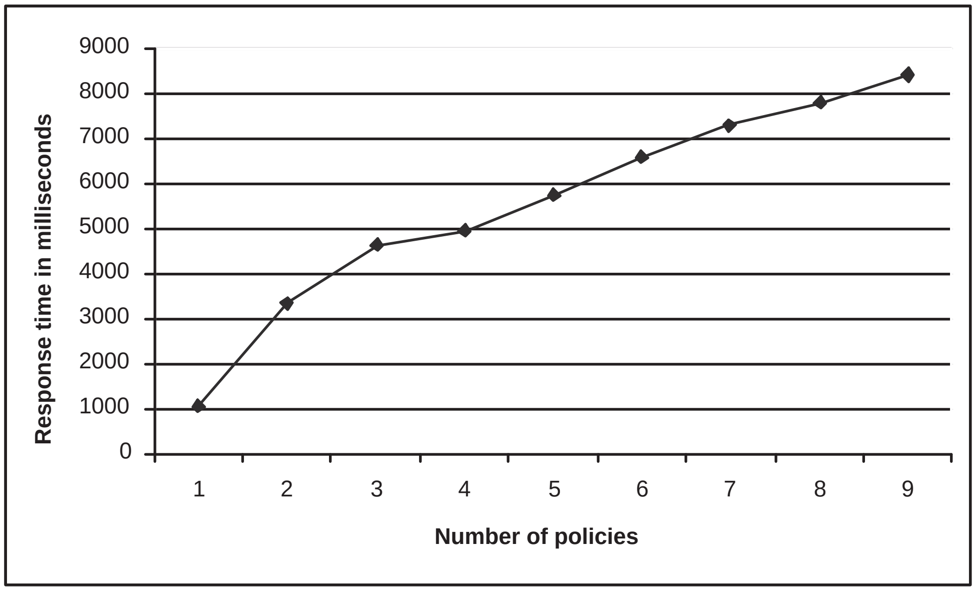


Figure 5.5: Polaris purchase card response time

5.5.2 Purchase Card Response Time

To measure the on-card performance of the Polaris purchase card system we tested the response time of the system while varying the number of active policy applets on a card. Figure 5.5 shows the time between sending transaction request data to the card and receiving a response[2]. The figure shows that the response time is roughly proportional to the number of active policies. We feel that a response time of under 10 seconds is acceptable for purchase card transactions, meaning that our prototype could support nine simultaneous policies with an acceptable runtime—consumers are used to waiting for a credit card authorization. However, something that a consumer would not notice—a delay of one or two seconds at most—would be better in that it would allow quicker checkout times and reduce the likelihood of annoying the consumer.

We think the response time of our system could be reduced with more effort. The defeasible logic engine is has been optimized to use EEPROM instead of RAM. We think a moderate use of RAM could speed up the processing while still keeping within the small RAM space available on the card. Our inference algorithm has not been optimized for the resolution function; it will check the provability of literals that do not impact the provability of the special yes literal. A more specialized inference algorithm could yield better response times.

Faster response times would allow Polaris to be used in applications like public transit systems where a transaction is no longer than the time needed to swipe a card through a card reader. A project that investigated the use of smartcards for access to the Japanese rail system set 100 milliseconds as the required maximum response time [67]. Other applications would also require faster response times. For example, the EZ Pass system

[1]All the on-card experimental results were performed using an Oberthur GalactIC Java Card. The class files were produced using Sun JDK1.4.2_05 javac compiler. The CAP files were generated using the Oberthur Comsopolic converter. Other converters may yield CAP files of other sizes.

[2]The time was measured using the IBM JCOP Java Card command shell. The policy applets were activated in a random order. The nine policies were the five from Figure 4.5, PC6 and PC12 from Figures 4.10 and 4.10, and two additional policies similar to the others.

equips vehicles with smart cards that communicate with roadside antennas to gather toll-payment and traffic information. If a car is traveling at 6.7 meters per second (about 15 miles per hour) then a 6 meter toll booth equipped with a short range wireless detector would need a response within a second. If a Polaris system was used to guard access to an EZ Pass account then the response time would have to be under 1000 milliseconds.

5.5.3 Code Generation

We measured the execution time of the Polaris code generator on a variety of policy models. We chose three policy models of realistic policies: *r3* has three policies enforcing a transaction limit, approved merchants and a five-purchase limit; *r5* is the example from Figure 4.5; and *r10* is r5 combined with r3 and policies PC6 and PC12 from the Penn Purchase Card policies. To stress our tools we also created artificial policy models: *gn* consists of n policies, each containing n modes each with its own randomly generated vote. The randomly generated votes consists of one defeasible logic rule with 0-4 antecedents. We randomly choose between strict, defeasible and defeater rules. Literals are selected randomly from a set of 27 literals, one of which is the special literal yes. All random distributions are uniform. We wrote the generated Java files to buffers in memory to minimize the cost of interacting with the file system [3].

Figures 5.6 and 5.7 show the time needed to convert the policy model to Java. Our tool ran out of memory when we tried to load the g90 model so we could not test the code generator on anything larger than the g80. For realistic policies the code generation time (which includes type checking) is very acceptable at under one second. Even for very large policy models like g70 and g80, which by file size is more than 100 times the size of the r5 model, the code generation time is under 15 seconds. In Figures 5.8 and 5.9 we compare the execution time to the size of the model, as measured by the size of the XML files used to store the model. For example, the r5 model from Figure 4.5 is 14 kilobytes, while the

[3]All the off-card execution time experiments were carried out on a 542MHz (as reported by Windows XP) Pentium III running Windows XP Professional, with 256MB RAM. We used the Sun Java HotSpot Client VM v1.5.0-b64).

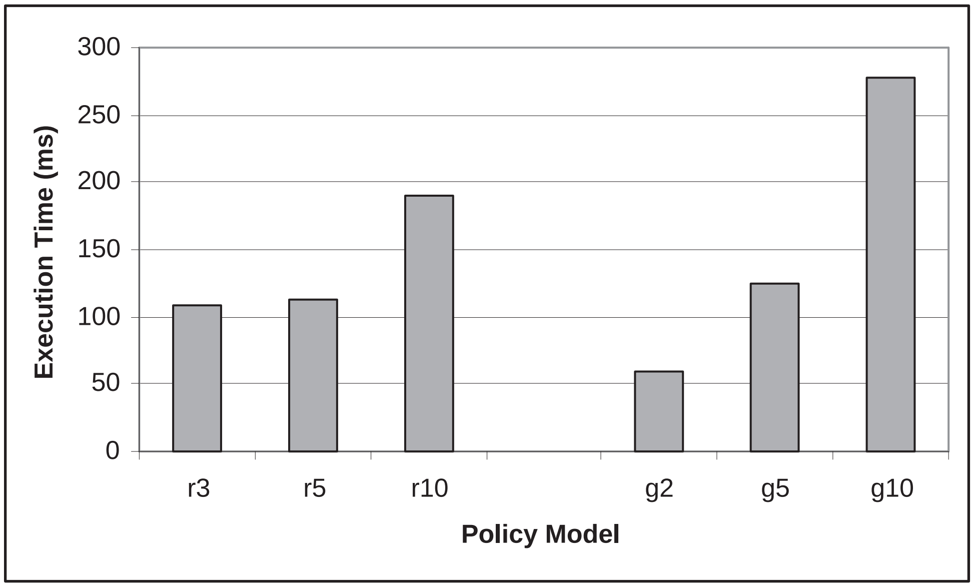


Figure 5.6: Code generation performance on small models

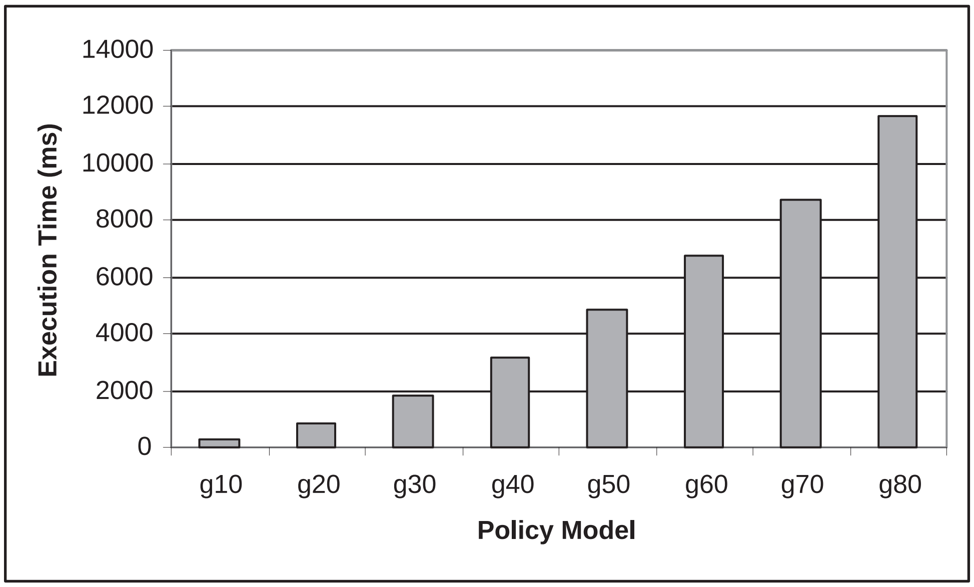


Figure 5.7: Code generation performance on large models

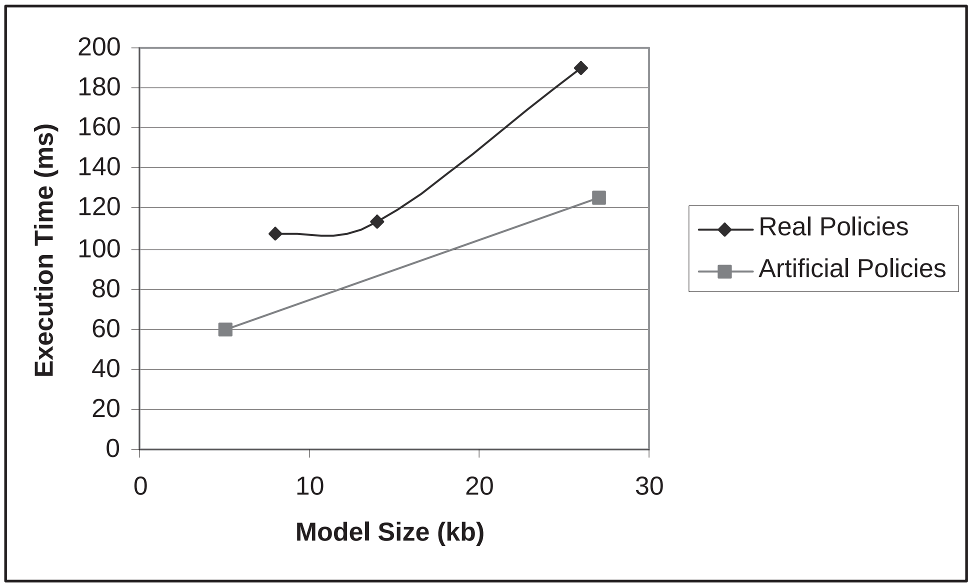


Figure 5.8: Code generation performance as a function of model size on small models

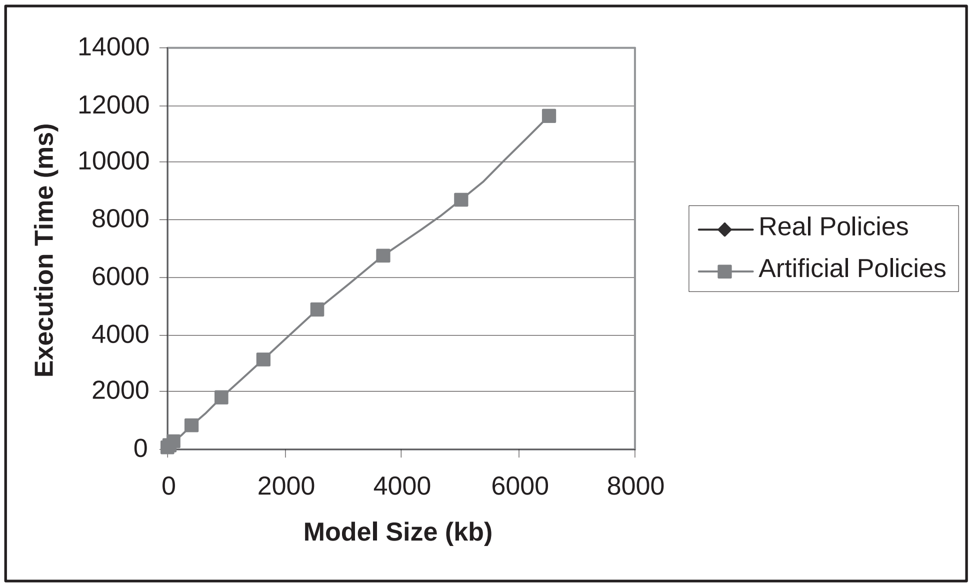


Figure 5.9: Code generation performance as a function model of size on all models

Policy Model	num. of policies	num. of vote combinations	time (ms)
r3	3	2	5
r5	5	16	43
r10	10	16	571
g2	2	4	5
g3	3	27	75
g4	4	256	1640
g5	5	3125	31130

Table 5.3: Conflict checking execution time for various policy models.

g80 model is 6500 kilobytes. The code generation time is essentially proportional to the size of the model for both the realistic and artificial models, as shown in Figures 5.8 and 5.9.

5.5.4 Conflict Detection

We performed similar experiments to measure the conservative conflict detection algorithm discussed in Section 5.2. This algorithm examines all possible combination of votes to check for conflicting combinations. It is conservative since it does not use reachability information to ignore combinations of votes which will never occur in a running policy model.

We restricted our measurements to smaller models as the algorithm is less scalable than the code generation algorithm. Again we measured our three realistic policy models and we measured artificial policy models g2, g3, g4 and g5. Table 5.3 shows the results of our experiments. The columns list the number of policy automata in each model, the number of distinct vote combinations that are possible in each model, and the time required to perform the conflict analysis. While we were able to analyze all our realistic models in under a second, our artificial examples are more demanding—g5 required more than 31 seconds of execution time. This difference in performance is due to the different characteristics of the real and artificial models. In our real policies many automata submit votes that are identical to other automata—our algorithm can ignore any duplicate votes since they will not affect the result of resolution function. In contrast, the artificial models usually all submit unique votes, and each automaton has many possible votes it could submit. This is reflected in the fourth column, which shows the number of vote combinations in the artificial models growing at a cubic rate with respect to the number of policies, while the realistic policy models do not have more than 16 distinct vote combinations.

5.6 Summary

In this chapter we presented the Polaris suite of tools—a prototype model-based design framework. Polaris includes tools to edit and analyze policy automata. Once a designer is satisfied with a policy design, Polaris will generate Java source code that can be compiled and run on a Java Card in a form that maintains the modular structure of the policy model; policy applets derived from automata can be added to a card dynamically in order to customize a purchase card.

Our experimental results demonstrate the feasibility of our framework. Applets derived from our policy automata occupy on average under 700 bytes of space on a card, allowing us to store more than 30 on a typical multi-application card. A Polaris-equipped Java Card is capable of enforcing up to 9 policies while maintaining response times under 10 seconds. Our code generation algorithm is capable of processing policy sets that are much larger than the size of the the Penn Procard policy set, and code generation for realistic examples was well under a second and thus presents no obstacle for a policy designer. Our conservative conflict detection algorithm was also able to handle realistic policy sets and terminate within one second, making it easy for a policy developer to check her policy before compilation.

Chapter 6

Security

In this chapter we discuss some security issues related to our model and implementation of a programmable purchase card. These issues include our assumptions about the environment and adversaries, which attacks we aim to prevent, which attacks the platform or payment infrastructure is responsible for preventing, and which attacks we cannot prevent.

6.1 Trust Relationships

One unusual aspect of our application that we only partially trust the cardholder. While we certainly would like to write policies that reduce the cost of a purchase card being stolen and used by an unauthorized (and therefore completely untrusted) individual, many of the policies in our examples deal with cardholders who should have partial but not complete access to the resources protected by the card. In our case, the policy developers and card issuers are trusted to determine what transactions are and are not permissible, while the user is expected to occasionally violate these policies—hence the need for enforcement by the card itself. The policy enforcer on the card acts on behalf of the secondary issuer, not the cardholder. However, we do not view the cardholder strictly as an adversary—there is some obligation to provide as much service to a valid cardholder as possible within the constraints of policies (this obligation corresponds to the principles of transparency and minimality in Section 3.1.3).

We must make certain assumptions to consider our programmable purchase card secure. Chief among these is we must trust the party—typically it is the merchant—who is supplying the transaction data to give accurate information; if the cardholder can conspire with a merchant so that an alcohol purchase is portrayed as a more benign item then there is little a purchase card can do to prevent abuse of the card. We justify this assumption partially by assuming that our payment card uses the existing financial infrastructure, where payment card issuers like banks and credit cards, in concert with law enforcement, make some effort to punish merchants who commit fraud.

This dependence on the merchant is exacerbated by the limited capabilities of smartcards. Most smartcards do not have an internal clock so any information about a transaction's time and date must come from the merchant. The merchant may conspire with the cardholder to manipulate the time reported to the card, perhaps avoiding time-based restrictions like the Penn Procard's policy that limits a card to 25 transactions per day.

Time is just one example of dynamic information that we would like to use to make policy decisions. Another is facts about who has been dropped from a list of approved suppliers. Similarly, a smartcard might wish to disable itself when an employee is terminated—a capability that would allow it to enforce the Procard policy PC2 in Section 4.4.2. As with time information, this data could be gathered by the merchant at the time of purchase and passed to the card, but this increases our dependence on the merchant (who, after all, has little incentive to inform the card that he is no longer approved). Gathering relevant information may also put a high burden on the merchant, who must somehow contact the appropriate databases for a particular set of policies.

We can use a secure signature scheme to reduce our dependence on the merchant. Many smartcards are capable of performing cryptographic operations, so it is feasible to demand information supplied by the merchant be signed by a trusted third party. Lyubich [46] gives a protocol for getting trusted time information using such a mechanism.

However, we still must depend on the merchant to gather such information, as the smartcard has no network connectivity other than what is supplied by the machine that it is communicating with. As mentioned above, gathering this signed data may be burdensome—a merchant may have a standard time server to get signed data about the current time, but it would be much more complex to fetch signed information from each university or company that does business with the merchant.

The limitations described above can be mitigated to a degree by assuming that our purchase card is one element of a layered policy enforcement system. In other words, we offer an addition to the existing payment infrastructure, not a replacement for it. For example, a purchase card transaction will still be subject to checks by banks and credit card companies to see if a given card has been revoked.

6.2 Attacks Using the Smartcard Platform

In addition to assuming accurate information, we must also assume the Java Card platform will behave as designed. In particular, we need to assume that an applet installed on the card will not have access to the private data in our manager applet or policy applets. The Java Card uses a firewall mechanism [65, 66] to enforce this, but if this system is compromised then an attacker could modify the manager applet or policy applet to allow previously forbidden transactions, or deny all transactions—for example, an attacker could modify the P_3 policy from Figure 4.5 so that it always remains in the state where it thinks three transactions have occurred and therefore no more should be permitted.

Our ability to dynamically add policy applets opens another avenue of attack. A cardholder (who has ample access to the card) or a merchant (who may have access to the card at the point of sale) may attempt to install a malicious policy. Such a policy could unduly restrict transactions (for example, a merchant may want to prevent purchases at a competitor) or override existing restrictions (for example, a child may wish to override a spending limit policy installed by a parent). To prevent such attacks we rely on the

GlobalPlatform [20] procedures for installing and removing applets. Since policy applets are just like other Java Card applets, the security measures designed to protect cards from malicious installation or deletion of applets extend to policy applets. The GlobalPlatform allows a card issuer to require cryptographic signatures for any installation or deletion of applets. An applet can therefore only be installed by a party who knows the relevant secret keys. These cryptographic restrictions partially mirror the trust relationship described in the previous section—the key holders are the trusted card issuer and secondary issuers, while the cardholder lacks the necessary secret information. A secondary issuer like a parent or enterprise can keep the keys secret from the cardholder and therefore prevent installation of malicious applets or deletion of previously installed policy applets.

Our implementation takes further steps to prevent abuse—if a policy applet is somehow deleted the manager applet refuses to process any further transactions, effectively disabling the card. We could implement additional safeguards whereby a policy applet shares a secret with the manager applet and therefore cannot be replaced with another applet with the same name.

A more humdrum attack to our payment card involves removing power from the card mid-transaction—sometimes this loss of power is called a 'tear'. Since a smartcard is typically powered by the card reader, this attack is as simple as removing the card from the reader. In our current implementation the policy applets update their state to record an approval or rejection before the manager applet notifies the card reader that the transaction is approved. Therefore there is a brief amount of time where some of the policy applets have recorded the purchase but the purchase has not taken place. Removing the card at this point will leave the policy applets in an inaccurate state. The Java Card platform provides some support for simple atomic memory updates—for example, we can ensure that either all policy applets are updated or none are. This eliminates an attack where someone removes power in mid-update, leaving some policy applets updated while others are not. However, it does not eliminate a tear attack entirely—a careful adversary could cut power at just the time when we have committed all updates but the transaction approval (or disapproval) has not been conveyed to the merchant. Unfortunately the Java Card atomic update system will not allow atomic updates that span multiple calls to the card, so we cannot delay committing the updates until we receive some sort of confirmation from the merchant. Even if such updates were possible, an attacker could remove power from the card before the confirmation is conveyed, leaving the card in a state where the transaction has occurred but the policy applets have not recorded it. (Hubbers and Poll [31] give a technique for reasoning about tears in Java Card programs.)

6.3 Summary

The security of our system depends on various assumptions about the parties taking part in transactions and the technology used to implement our payment card. In particular, we trust the merchant to supply accurate transaction information, or to supply such information from a trusted party. We also trust the Java Card hardware and system to faithfully implement the protections described in the Java Card and GlobalPlatform specifications. We leverage the security assurances of the the Java Card platform and the established payment infrastructure to ensure that our system behaves securely.

Chapter 7

Conclusion

We have presented a thorough examination of a programmable payment card—a smart-card capable of holding and enforcing multiple modular purchasing policies. The application has been explored from a formal perspective by proposing a succinct formal model for modular policies called policy automata. Building on this formal model, we defined properties of policies and algorithms for checking these properties.

We also demonstrated how this formal model can be part of an effective model-based design framework. The effectiveness of the model was demonstrated by showing how it can encode the real world purchasing policies of the University of Pennsylvania. The feasibility of our model was demonstrated by implementing an editor, an analysis tool, and a code generator which can translate the formal description of policies into executable Java Card applets. Our experiments show that the restricted resources of a Java Card do not prevent our policies from giving response times that are acceptable for real world purchasing situations.

7.1 Open Issues and Future Work

There are a number of open issues that offer directions for future investigation.

One research direction related to the formal model of the application deals with reject-observing policies like the ATM policy discussed in Section 3.1.5. Recall that this policy disables a card after three failed attempts to make purchases that violate a policy. Such a policy is actually used for bank cards, so it would be nice to use our formal model to analyze it. It is possible to encode the policy as a policy automaton. However, the formal definition of 'security policy' discussed in Section 3.1.1 does not distinguish the ATM policy from a policy that blocks bad purchases without ever disabling the card. It is not clear if there is a modification of the security policy definition that retains the simplicity of the current security policy framework.

We would like to further investigate different notions of refinement in the hopes of finding a definition that is both succinct and appropriate for policy automata. Ideally, such a definition would allow us to characterize the composition of policy automata formally without referring to the operational semantics of the automata. With truncation automata we know that the composition of two automata will enforce the conjunction of the corresponding policies—we would like to make a similar statement for policy automata.

Section 5.4 describes adding policy automata to cards dynamically. The implementation allows a user to install some policies, make some purchases, and then add more policies. However, the formal semantics assumes that all policies are installed before any purchases are made. We would like to extend the formal semantics and the algorithms for checking automata properties to account for the possibility of adding new policy automata in the middle of a transaction sequence.

In the current formal model any policy can submit any vote. However, it may be useful to have restrictions on what votes a given policy may be able to submit. For example, in our example in Section 4.2, the P_E policy signaled that a transaction request is an emergency transaction by submitting a vote "$\{\} \rightarrow e$", which modified the effect of P_N's vote by asserting that the literal e is true. However, any policy could submit the same vote, effectively fooling P_N into thinking that P_E had marked the transaction as an emergency. If we added some restriction that allowed only P_E to submit votes that imply e then P_N

could be sure that if e was marked as true then it was a genuine emergency as decided by P_E.

Similarly we may want to restrict which policies can override other policies. For example, the Penn Procard policy PC9 sets a $5000 per month limit on purchases, but this limit can be raised with approval from a senior financial officer. We can override a policy automaton implementing PC9 by installing a new policy that submits votes that override the votes of the PC9 automaton—however, we may want to ensure somehow that only policies approved by the senior officer could submit such a vote.

Many formal trust management systems [38, 36, 40] have been proposed that deal with delegation as a mechanism for authorization. The programmable payment card enables delegation by giving enterprises or other parties a way to customize a payment card and then delegate it to another party. However, this delegation is external to the formal model of policies discussed in Chapter 3. We are interested in investigating the connection (if any) between the kind of delegation whereby a secondary issuer gives a customized card to a cardholder, and the existing formal systems for describing delegation in trust management. Perhaps this direction of research could be combined with some mechanism for restricting which policy automata can submit certain votes, as described above.

This dissertation discusses using defeasible logic statements as votes. However, it is likely that the people who write real world purchasing policies will not be familiar with such a formal notation and cannot invest the time to master defeasible logic. Instead, it would be desirable to have some a simpler language for writing votes, with a semantics based on defeasible logic. Such a language could be based on simple English language templates which are translated to defeasible logic; for example, a policy designer may write “tentative yes if not an emergency”, which would then be translated to “$\neg e \Rightarrow$ yes”. Halpern and Weissman use a similar approach for describing polices in Lithium [24].

As discussed in Chapter 6, policies may need access to data which cannot be kept on a payment card. For example, a policy that blocks purchases while an employee is suspended must somehow learn of the suspension. Chapter 6 sketched some possible techniques for using such data; in the future we would like to flesh out these ideas, adapt our formal model if necessary, and implement a system that securely uses off-card data for policy decisions.

Our implementation can be improved in a number of ways. Further optimization of the manager applet on the card, especially the defeasible logic engine, is necessary to achieve transaction processing that is fast enough to handle dozens of policies, or to make access control decisions in domains that require response times under a second—for example, if a cardholder engages in a transaction by walking or driving past a wireless card reader. One possible strategy for optimization is to use some sort of incremental compilation, where defeasible logic votes are 'compiled' when a policy is installed so that when the votes are resolved a simple table-lookup is used instead of running a defeasible logic inference algorithm.

The analysis tool currently implements a conservative conflict checking algorithm. In the future we would like to implement the full conflict checking algorithm which makes use of reachability information. Additionally, we would like to implement algorithms for checking redundancy (both normal and strong).

Finally, while we focused on the programmable payment card application in this work we think the policy automata framework could be applied in other domains. Some preliminary work has been carried out at the University of Pennsylvania Security Lab that applies our techniques to regulating mobile phone use. A project at the University of Illinois at Urbana-Champaign has applied some of our techniques to managing policies for web services. We are interested in continuing this direction, perhaps extending the formal model or defining new properties if the new applications warrant.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[2] Rajeev Alur, Radu Grosu, and Michael McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 280–295, 2000.

[3] Rajeev Alur, Michael McDougall, and Zijiang 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.

[4] Grigoris Antoniou, David Billington, and Michael 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.

[5] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[6] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[7] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[8] David Billington. Defeasible logic is stable. *Journal of Logic and Computation*, 3(4):379–400, 1993.

[9] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

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

[11] Gerhard Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[13] Marco Cadoli and Marco Schaerf. A survey of complexity results for nonmonotonic logics. *Journal of Logic Programming*, 17(2/3&4):127–160, 1993.

[14] 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.

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

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

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

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[19] Philip W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, pages 43–55, 2004.

[20] GlobalPlatform. *GlobalPlatform Card Specification v2.1.1*, March 2003.

[21] Benjamin N. Grosof, Yannis Labrou, and Hoi 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.

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

[23] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. Technical report, The MITRE Corporation, 1997.

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

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

[26] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119. ACM Press, 2000.

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

[28] James A. Hoagland, Raju Pandey, and Karl N. Levitt. Security policy specification using a graphical approach. Technical Report CSE-98-3, University of California, Davis Department of Computer Science, 1998.

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

[30] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[31] E.-M.G.M. Hubbers and E. Poll. Reasoning about card tears and transactions in Java Card. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *LNCS*, pages 114–128. Springer-Verlag, 2004.

[32] Nijmeegs Instituut Voor Informatica En Informatiekunde. Esc/java 2. `http://www.cs.ru.nl/sos/research/escjava/`.

[33] International Organization for Standardization. *ISO 18245:2003 Retail financial services – Merchant category codes*, April 2003.

[34] Bart Jacobs, Hans Meijer, and Erik Poll. VerifiCard: A European project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001.

[35] Java compiler compiler. `http://javacc.dev.java.net/`.

[36] Trevor Jim. Sd3: A trust management system with certified evaluation. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 106. IEEE Computer Society, 2001.

[37] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[38] Butler Lampson and Ron Rivest. SDSI—a simple distributed security infrastructure. `http://theory.lcs.mit.edu/˜cis/sdsi.html`.

[39] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[40] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003.

[41] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2004. To appear.

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

[43] M. Lyubich. Eine SET Kundembörse mit der Java Card Unterstützung. In *GI Informatiktage 2000*. Konradin-Verlag, November 2000.

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

[45] M. Lyubich and C. Cap. Eine implementierung von SET für Java. In *Tagesband Netzinfrustruckhur und Anwendugen für Informationsgesellschaft*, pages 208–214. Dr. Wilke Verlag, 1998.

[46] Mykhailo Lyubich. *Architectural Concepts for Java Card Running a Payment Protocol and Their Application in a SET Wallet*. PhD thesis, University of Rostock, 2003.

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

[48] Michael J. Maher, Andrew Rock, Grigoris Antoniou, David Billington, and Tristan Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001.

[49] Mastercard and Visa. *SET Secure Electronic Transaction Specification: Business Description*, May 1997.

[50] Mastercard and Visa. *SET Secure Electronic Transaction Specification: External Interface Guide*, May 1997.

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

[52] Mastercard and Visa. *SET Secure Electronic Transaction Specification: Programmer's Guide*, May 1997.

[53] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.

[54] Sun Microsystems. Java card platform. `http://java.sun.com/products/javacard/`.

[55] Donald Nute. Defeasible reasoning. In *Proc. 20th Hawaii International Conference on Systems Science*, pages 470–477. IEEE Press, 1987.

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

[57] University of Pennsylvania. Transaction compliance audits. `http://www.purchasing.upenn.edu/about/pm_audit.php`.

[58] D. Pool. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[59] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 12(1-2):81–132, 1980.

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

[61] Saheem Siddiqi and Joanne 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.

[62] Medical Economics Staff. *Physicians' Desk Reference*. Thomson Healthcare, 57 edition, 2003.

[63] Scott D. Stoller and Yanhong A. Liu. Security policy languages and enforcement. In *Proceedings of the Third Russian National Conference on Mathematics and Information Technology Security (MaBIT-04)*, October 2004.

[64] Sun Microsystems. *Java Card 2.2 Application Programming Interface*, September 2002.

[65] Sun Microsystems. *Java Card 2.2 Runtime Environment (JCRE) Specification*, June 2002.

[66] Sun Microsystems. *Java Card 2.2 Virtual Machine Specification*, June 2002.

[67] Hideyuki Tokuda, September 2004. Keynote address at EMSOFT 2004, Pisa, Italy.

[68] Kansas State University. Bandera project. `http://bandera.projects.cis.ksu.edu`.

[69] University of Pennsylvania. *University of Pennsylvania Purchasing Card Cardholder Guide*, April 2003.

[70] Joachim van den Berg and Bart Jacobs. The loop compiler for java and jml. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312. Springer-Verlag, 2001.

[71] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[72] Avishai Wool. Architecting the lumeta firewall analyzer. In *10th USENIX Security Symposium*, pages 85–97, Washington D.C., August 2001.