ABSTRACT

IRWIN, KEITH A System for Managing User Obligations. (Under the direction of Assistant Professor Ting Yu).

As computer systems become a more pervasive part of our societies, actions within those computer systems are becoming increasingly governed by complex policies such as laws, corporate policies, and legal agreements such as data sharing agreements and privacy policies. These policies impose both requirements about what may or may not be done and about what must be done. Current security policies may be able to manage restrictions on actions, but they are not sufficient to describe actions which are required. We examine herein the idea of user obligations, which are actions which are required of the users, but which the system cannot directly cause to occur.

We propose a system for the management of user obligations. This system should both ensure that obligations are assigned in a manner such that it will be possible for them to be fulfilled and allow users of a system to know what they are required to do. We present an abstract formal model of such a system. We examine a number of aspects of such a system, principally including the maintenance of an acceptable system state, the assignment of blame when users fail to fulfill their obligations, and providing adequate feedback to users when their actions are rejected. For each of these aspects, we present formal definitions to define the range of acceptable behavior.

We also provide a more specific and concrete model of one possible user obligations management system and develop algorithms for that model. We do this in order to show the practicality of our formal models and properties.

A System for Managing User Obligations

by
Keith Irwin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Peng Ning	Dr. Munindar P. Singh
Dr. Ting Yu Chair of Advisory Committee	Dr. S. Purushothaman Iyer Co-Chair of Advisory Committee

DEDICATION

To my beloved wife Joy, without whom this would have been much more difficult.

BIOGRAPHY

Keith Irwin received his BS in Computer Science from Carnegie Mellon University in 1998.

ACKNOWLEDGMENTS

First, I would like very much to thank my advisor, Dr. Ting Yu, for all of his help. His aid in directing the research has been invaluable, and from him I have learned a great deal about effectively framing problems and presenting research.

Second, I would like to thank Dr. William Winsborough, our collaborator on all of the research which is contained in this thesis. Dr. Winsborough has been supportive throughout the process and has offered many helpful insights into the research.

Third, I would like to thank all of my friends and fellow students who have helped me out. Travis Breaux was a great person to talk to regarding the ideas in the thesis. Emre Can Sezer was a big help when I needed someone to help get me moving. And I would also like to thank all of the students in our research group, Dr. Anton's research group, Dr. Ning's research group, and Dr. Reeves' research group who have helped me by workshopping papers or listening to practice talks.

Fourth, I would like to thank my friend Robert Cauthen for helping to proofread my dissertation and in other ways being a good friend.

Lastly, I would like to thank Dr. Yan Solihin, my first advisor. Although we wound up not continuing to do research together, I learned quite a lot from him.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In recent years we have seen a marked increase in the amount of laws, regulations, agreements, and policies which govern computer systems. For instance, privacy laws and data-retention laws adopted recently in several countries directly regulate the processing of data in computer systems. In the United States, for example, HIPAA (the Health Insurance Portability and Accountability Act) is a recently adopted health-care law which imposes regulations on the handling of private health-care data in information systems. Additionally, many laws have an indirect effect on computer systems. For example, the Sarbanes-Oxley laws in the United States, which govern the accounting and reporting practices of publically-held corporations, have resulted in many firms having to change their computer systems. Although the law does not make any specific requirements of computer systems, many corporations use computer systems for their accounting and reporting, so they have had to make significant adjustments to their computer systems as a result of the act.

We have also seen an increase in legal agreements which govern computer systems. The most common of these are privacy policies. A privacy policy is an agreement by a data-collector to handle private data in a specific manner. Most large commercial web sites have privacy policies which govern their data usage. In many cases, collected data may be shared, but the partners with which the information is shared are also governed by the privacy policy under which the data was collected. Privacy policies primarily govern issues such as what the data will be used for. However, some also contain provisions stating when the information will be erased or requiring notification to the user in certain circumstances.

Many of the policies which now make requirements of computer systems describe

what the systems are and are not allowed to do. Those sort of requirements can usually be implemented by creating appropriate security policies, and there exists quite a bit of literature describing security policies, how to enforce them, and how to analyze them.

However, many policies also dictate what must be done in a computer system. For instance, some privacy policies require that data be deleted after a certain amount of time. Also, under HIPAA, if a patient makes a request to a health-care provider for information about what data the health-care provider has stored about the user, then the provider is required to respond. Also, other legal agreements, such as data-sharing agreements, often make requirements of computer systems.

These requirements go beyond what security policies can model. As such, we want to describe a new type of computer system policy which helps bridge the gap between what outside policies require and what computer systems enforce. We refer to requirements of positive action as *obligations*. If a system must ensure that some task is carried out then that task is an obligation of the system. There are certainly at least some obligations which a system would have which can be carried out directly by the system itself. These can be implemented through some form of computer program.

However, many obligations which a system might have will need to be carried out not by the system itself, but rather by its users. For instance, the reporting and accounting requirements imposed by Sarbanes-Oxley require that certain corporate officers sign off on certain reports. Clearly this process cannot be carried out by a computer program alone. Instead we suggest that in many cases, a system's obligation can be broken down into a set of obligations which are then given to the users of the system. That is, there could be a set of actions which the users of the system must carry out in order to fulfill the system's obligation. For instance, if a patient makes a request for information from their health care provider, this might result in an obligation being assigned to a particular employee of the health care provider to find the required information.

We refer to these as user obligations. User obligations may also arise out of policies which govern the actions of users directly. For instance, a user who makes a change to a sensitive file might be required to document that change or a user who checks out a book might be required to check it back in within some time period.

Speaking generally, a user obligation is simply a requirement that a given user take some action within some future time frame. Most people already regularly incur obligations

to take actions due to organizational policies. Currently, most of these actions do not take place within the context of a computer system. If current trends continue we can expect that more and more of the actions we take will involve computer systems. This means that computer systems will have the opportunity to manage user obligations.

As computer systems become more integrated into greater and greater portions of our societies, the number of policies which governs computer systems is likely to grow. As such the number of system obligations is likely to grow, and this, in turn, will increase the number of user obligations. There are several problems which are likely to arise as the number of user obligations rises. As such, we propose that it is desireable, perhaps even necessary, that computer systems which can manage their user obligations be created. Building computer systems which manage user obligations can help deal with, or at least mitigate, a number of problems.

The first problem is a simple one: a user may not know his obligations. Obviously, a prerequisite to any reasonable system which involves user obligations is that the system inform the users of their obligations. But, as users are involved in organizations and computer systems of increasing scope and complexity, it becomes more and more difficult for users to keep track of all of their obligations. As such, the system could keep track of this information for them. Rather than having to know what obligations they have pending, a user could simply consult the user obligations management system to find out his obligations.

The second issue is the question of whether or not obligations are being assigned in such a way that they can be fulfilled. User actions are governed by security policies which state what the user can and cannot do. If our obligation policy assigns an obligation to a user to do some action, but the security policy prevents that user from taking that action, then we clearly have a problem. Also, there is the possibility of user obligations which directly or indirectly interfere with one another.

A third issue is how blame for obligation failures is assigned. Because obligations require user action, there is a possibility that they will go unfulfilled. However, obligations given to different users may be related. For example, one user might be required to submit a report and a second user would be required to approve the report. When obligations go unfilled it could simply be that a user chose not to fulfill their obligation, but it could also be an obligation was not fulfilled because it could not be fulfilled due to one or more

previous obligation failures. As such, managing obligations should include some means of discovering who is to blame for failures when they occur.

A fourth issue is understanding access control decisions. Because user obligations express system goals, it will be important in managing them to prevent users from interfering with the obligations of other users. However, because obligations are temporary in nature, they are going to be difficult for users to predict. And as the number of user obligations grows, they may interact in complex ways, resulting in access control decisions being difficult for users to understand. As a result, we have an added burden of helping the user understand and deal with access control restrictions.

All of these tasks may be accomplished by a well designed user obligation management system. In the next chapter, we will give a description of how such a system would operate, describing both the basic components and a general description of the policies which govern the operation of the different components. In the next section, we describe a formal model of the system which will be used to prove results about the system and its components. Following that, we describe several particular components of the system in greater depth.

1.1 Background

What we are attempting to do in this dissertation is to describe and formally model a system for managing user obligations. There is a fair amount of work which is in some way related to this work, but very little immediately germane to solving the problems related to such a system.

When it comes to the general idea of obligations, there has been a fair amount of work which has in some way or another touched on obligations. There appears to be a growing belief that obligations of a system are important things which should be dealt with or modeled in some way. For example, a number of different policy languages now model obligations. Some languages, such as XACML [49] and KAoS [51] model obligations as being tasks which must be carried out by the reference monitor when certain access control decisions are made. This is adequate for a limited range of system obligations, but obviously not helpful for modeling user obligations.

Ponder [12] and Rei [25] both support the specification of user obligations. How-

ever, in the basic constructs of both languages, time constraints of obligations, such as deadlines, cannot be directly expressed. As such, it is difficult, if not impossible, to express obligations in Ponder or Rei which can be violated.

Bettini et al. [5] studied the problem of choosing appropriate policy rules to minimize the provisions and obligations that a user receives in order to take certain actions. In their policy model, each privilege inference rule is associated with a set of provisions (actions that have to be taken *before* a request can be granted) and obligations. Unlike the policy model used in this paper, they assume actions in provisions and obligations are disjoint from those requiring privileges. In other words, they can always be fulfilled. This sort of assumption avoids most of the questions that we seek to address, but is unlikely to be a realistic assumption in a real-world system since user actions are almost always governed by security policies. In later papers [6, 7], they also dealt with issues related to detecting when obligation failure has occurred and how to deal with violations. However, they still kept their assumption that obligated actions would not be prevented by security policies, thereby allowing them to assume that a user who failed an obligation was always to blame for that failure.

A work which tackles a similar problem to ours is Gama and Ferreira's paper about Heimdall [15], a prototype obligation monitoring platform which keeps track of pending obligations. The focus of this paper is on detecting when obligations are fulfilled or violated. This requires the modeling of time constraints in obligations, which are explicitly supported in its policy language, xSPL. As such, their model of obligations resembles our more than any other policy language we have found. In this dissertation we avoid retreading the ground which they have covered, instead refering to their paper to understand how compliance monitoring could be carried out.

There have also been a number of works which focus on obligations between two parties in a web services or grid services settings. In these sort of situations, we are generally looking at obligations between one distinct organization and another. This scenario is distinct from user obligations because security policies are not directly involved and it is not reasonable to have a central authority which can enforce an operational policy in their situation. One example of this is Sailer and Morciniec [41] who propose a means of using a third party to monitor obligation compliance in contracts in a web services setting.

This dissertation tackles several different research problems related to a user obli-

gations management system. Rather than go into the background of each problem at this point, we save discussing other works related to each one for the different chapters which cover particular problems.

Chapter 2

User Obligation Management System

Before we discuss the specifics of how the user obligation management system works, we are going to describe the different objects of the system. First are actions, which are things which users do in the computer system. Second are user obligations, which are requirements that users do certain things. Third are events, which are things which have an effect on the computer system. Actions are a special type of event. Specifically, they are events which are initiated by users. There may also be events which originate from outside the computer system and are not user initiated.

The design of the user obligation management system is based on an event-driven model, the basics of which we have just described, but which we will formalize in chapter 3.

2.1 Actions

When we discuss actions in the context of a user obligation management system, we have a fairly broad idea of what a user action can be. In existing computer systems, user actions are often things such as access to a file or queries to a database. However, when we use the term action in this thesis, we refer to pretty much any thing which a user can tell a computer system to do. The only constraint is that an action must not require additional input from the user. If there is some task such that the user does something and then the

computer system responds and then the user, in turn, responds to that, that task should be broken down into multiple actions. Also, a user action is something which the user must do of their own free will. If the computer system can force the user to take the action, then we do not consider it a user action, but rather a system action.

2.2 Obligations

When a user is required by the policies of the computer system to take some user action, we call that a user obligation. One property of user obligations is that they are not enforceable by the computer system. This is because user actions are voluntary actions. A policy may require some action, but it cannot directly force that action to occur. There may be consequences to the user for not carrying through on their obligations, but this is not the same as a requirement to take an action. If the computer system will carry out the action regardless of the actions of the user, then we do not model it as a user obligation.

A second property of user obligations is that they should be able to be monitored. Although the computer system cannot cause an obligation to be fulfilled, it should be able to know whether or not an obligation has been fulfilled. If it is impossible for a computer system to know whether or not a given obligation is fulfilled, then that obligation is not going to be relevant to the system's decisions or analysis. A reasonable policy might still assign such obligations in at least some cases, since it may be the case that there are needed actions which are outside of the immediate scope of the computer system, however we will refrain from modeling these obligations in our user obligation management system.

We model obligations as being a tuple of a user, an action, and timeframe in which that action must be carried out by that user. We model the timeframe as a simple pair of a start time and an end time. In particular, as we will go into later, in this system, all obligations are modeled as being event-driven, which is to say that obligations are incurred as a result of events which happen in the system. Note that this can include events which are caused by user actions and that one possible user action would be to request to be assigned an obligation or to request that someone else be given an obligation. So even though obligations all come from events, in practice they will often come from the users.

2.3 Events

Quite simply, an event is anything which happens which changes the state of the computer system. We divide events into two categories: controllable events and uncontrollable events. User actions are controllable events. In our user obligation management system, as in many systems, if a user attempts to take an action, this action may be allowed or disallowed depending on the security policy under which the computer system operates. If a user action is allowed, then that action becomes an event. Because these events can be vetoed by the system they are considered controllable events.

Uncontrollable events are events which are outside the control of the computer system. These can include events which come from entities outside the system, such as customer orders or audit requests. Also, this category includes events which are not generated by any entity, such as hard drive failures, power outages, or Ragnarok. In essence, an uncontrollable event is anything which will effect the state of the computer system when it happens, but that the system cannot veto.

2.4 User Obligation Management System Overview

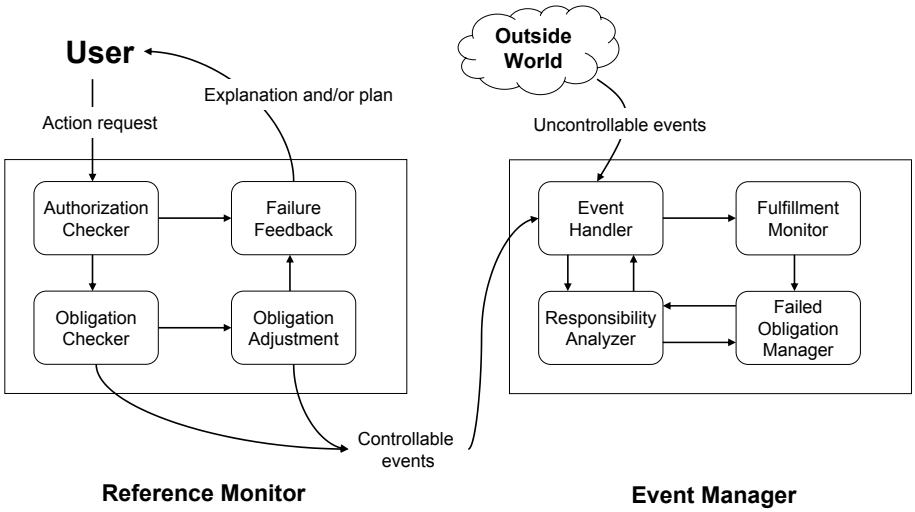

Figure 2.1: The major components of an obligation management system

Figure 2.4 shows the basic structure of a user obligation management system.

There are two primary components: the reference monitor and the event manager. As in traditional access control systems, the reference monitor makes decisions related to user actions. It can permit or deny the action that a user requests to perform. If a user action is permitted and successfully performed, it becomes an event which is passed to the event manager. It is passed to the event manager because such an event may trigger obligations. Events passed from the reference monitor are controllable events.

The event manager deals with both controllable and uncontrollable events. The event manager can add or update obligations in response to events and according to obligation policies. For some events, the event manager may even remove existing obligations. The event manager also monitors user-initiated events to see if existing obligations become satisfied or violated.

There is also a third optional component, the policy analyzer, in an obligation management subsystem. The policy analyzer performs static analysis of policies to check certain properties of policies, such as whether or not there exist situations where users will be given obligations which they can never fulfill. The policy analyzer is useful when a security officer designs the policies which govern access control and obligations. It does not interact with other system components during the operation of a system. Presently, the policy analyzer is not a component which we have studied in depth.

2.4.1 Reference Monitor

The reference monitor is composed of several sub-modules: the authorization checker, the obligation checker, the obligation adjustment module, and the failure feedback module.

Authorization Checker

When a user requests to take an action, the authorization checker decides according to the access control policy whether the user is authorized to take the action when the system is in its current state. The access control policy is simply a policy which describes what actions are permitted in what situation. The authorization checker is a direct analogue of the reference monitor in existing secure systems. Because the issues relating to implementation of modules to enforce security policies are well covered in the literature, we do not provide an

in-depth description here. Our system does not constrain the functioning of this module in any special way and access control policies written in any existing security policy language are acceptable. If the user action is rejected by the authorization checker, the action is simply denied and the obligation checker and obligation adjustment module are not involved.

Obligation Checker

If, on the other hand, the user request passes the authorization checker, it is necessary to further check that, if the action were performed, the resulting obligations would still keep the system in a secure state. This is the responsibility of the obligation checker module. This module is necessary because it is a goal of the user obligation management system to ensure that obligations can be fulfilled. This is not a trivial task due to the possibility of complex interactions between different obligations and between obligations and events. In particular, the obligations checker attempts to ensure that some operational policy can be met. One operational policy is that of *accountability*, which is the state such that if all users attempt to fulfill their obligations, then all users will succeed in fulfilling their obligations. Accountability is discussed in more depth and formally defined in section 3.2. The specific operation of an obligation checker is described in depth in chapter 4.

Obligation Adjustment

If the request also passes the examination of the obligation checker, then the action is allowed to occur, it becomes an event, and it is passed on to the event manager. Otherwise, the system will invoke the obligation adjustment module, which tries to make adjustments to existing obligations such that the altered obligations keep the system in a secure state while still allowing the requested action to happen.

For example, an obligation policy may require an obligation be fulfilled within a certain time frame. However, if we were to issue an obligation whose time frame is contained within the required timeframe, then the adjusted obligation would still be consistent with the obligation policy. Issuing a more constrained obligation may allow the set of obligations to be consistent with the operational policy in situations where an obligation with the full time frame would be incompatible with it.

As such, the obligation adjustment module considers whether there is a possible

set of obligations which can be altered in such a way as to permit the action, but still be consistent with both the obligation policy and the operational policy. Because this can involve not just modifying the obligation which will be issued in response to a given action, but perhaps also modifying other obligations which exist in the system, this module may also weigh the costs of the disruption of making changes to existing obligations when making its decision. If it successfully identifies such an acceptable adjustment, it will make the changes accordingly and the requested action will be allowed. In some cases, making adjustments will require communication with the events manager.

The specifics of obligation adjustment are not something which this dissertation currently covers, however, it is a topic for future research. As such, additional discussion of it may be found in section 8.1.

Failure Feedback

If, however, obligation adjustment fails, the user request should be rejected. Besides informing the user about this decision, the user obligation management system also invokes the failure feedback module. As greater number of user obligations occur in a system, it becomes increasingly difficult for users to understand the complex interactions between obligations. Thus we need to provide improved mechanisms to enable users to be able to find ways to reach their goals when their actions are denied. In some cases, we may be able to provide a simple explanation of how the user has run afoul of the access control policy. However, for cases where the user has run into problems with the operational policy, the situation is more complicated.

As such, we have described and implemented a failure feedback module which will make an attempt to find a plan which, if carried out, would allow the user to take the action which she wishes to. This plan may require the cooperation of multiple users. In some cases, no plan may be possible. The operation of this module is described in more detail in chapter 6.

2.4.2 Event Manager

The event manager takes events as its input. User-initiated events enter the event manager from the reference monitor. Uncontrollable events are considered to appear of

their own accord. An event is dealt with by several submodules of the event manager, including the event handler, the fulfillment monitor, the responsibility analyzer, and the failed obligation manager.

Event Handler

The event handler determines what should happen in response to an event according to the obligation policy. Where there are system actions (that is, actions done automatically rather than by users) that need to happen, such as changing the permission state or logging events, the event handler does them. If user actions are needed, the event handler creates and assigns appropriate obligations. User obligations will be derived from an obligation policy which dictates what obligations should be assigned in response to an event in what circumstances. This includes user-initiated events, such as when a patron successfully checks out a library book, they will be given an obligation to return it. And it also includes uncontrollable events, such as when a hard drive failure occurs, a system administrator will be given an obligation to replace it within twenty-four hours.

From a theoretical standpoint, the operation of this module is straightforward. It must simply match currently occurring events to events in the rules of the obligation policy and if the current state matches the condition from the given rule, then the specified obligations are added to the list of pending obligations. However, in practice, these steps may have some difficulties involved depending on the specifics of the systems and languages involved. We do not address those challenges in this work.

There is at least some existing research which deals with these sort of issues. Specifically, Heimdall [15] is a platform which handles the roles of the event manager and also the fulfillment monitor for platforms based on xSPL, an extension to the SPL language [38]. Their system is similar to the Event Manager portion of our proposed system, except that there is no blame assignment component to their system.

Fulfillment Monitor

The fulfillment monitor maintains a list of pending obligations. When a user-initiated event occurs, it determines whether any of the pending obligations is fulfilled by it. It also checks whether any pending obligations have time limits that have passed

without appropriate actions having occurred, thereby having been violated. If any have been violated, it notifies the failed obligation management module. Again, theoretically, the operation of such a module is quite straight-forward, and we do not go into depth about implementation specifics here.

Failed Obligation Manager

The failed obligation manager deals with the consequences of obligation failures. This can actually be quite a complex and varied task, the specifics of which will depend on the particulars of the system.

In most systems we would expect this to module to, at very least, report and record obligation failures. However, it also may be tasked with responding to obligation failures in one or more different ways. To begin with, certain obligation failures may carry consequences for the users who are responsible. For instance, a library patron which fails to return books by the deadline may be disallowed from checking out more books. Administering those consequences would be the job of this module. In most cases, it is quite clear which user is responsible, but in some other cases, especially when there are multiple failed obligations, the determination may be complex. In such a situation, the failed obligation management module may consult the responsibility analyzer.

Also, in some cases, the failure of certain obligations may result in obligations being given to other users to deal with the consequences of the failure. For example, if an IT employee fails to respond to a ticket within 24 hours, his boss may be given an obligation to look into it and file a report about what happened. This assignment of obligations would generally be carried out by the failed obligation manager.

In some systems, however, the failed obligation manager may also be called on to check whether or not the existing obligations are still in compliance with the operational policy. If the operational policy is being violated, it would then need to modify the state through obligation modification and/or system actions so that the system is brought back to a secure state. This process will, in many cases, use the same logic as is used in the obligation adjustment module in the reference monitor. However, it may even have additional leeway to do things in order to restore the system to a state of compliance with the operational policy. For instance, it may be allowed to discharge existing obligations or to add new obligations to the system. When there are several possible ways to change the system

to restore compliance with the operational policy, it should choose the least disruptive according to some reasonable heuristic.

Research on this module is connected to research on the obligation adjustment module and is described in section 8.1.

Responsibility Analyzer

The responsibility analyzer monitors incoming obligation-related events such as obligation assignments, completions, and failures (violations due to obligations not being fulfilled on time). It maintains the information needed to know who is responsible for what obligations, and who is responsible when an obligation is violated, including when one obligation going unfulfilled renders other obligated users unable to fulfill their obligations, directly or through a cascade of failures.

This module however, is not a postmortem analysis tool. Rather it maintains responsibility information in an on-line manner, adjusting it as circumstances dictate. Because responsibility is a more complex concept than simply examining which obligations satisfied which preconditions (as we argue in more depth in chapter 5), there is more than one possible assignment of responsibility for a given circumstance. As such, we use a responsibility policy to guide us in determining who is considered to be responsible for which other obligations.

If an obligation failure later occurs, then we use the responsibility information to assign blame. It is restricted in its blame assignments by the principle that if a user is definitely responsible for an obligation (a concept which we will formally define later), then the module should consider him to be responsible, and also by the principle that if a user is definitely not responsible then the module should not consider him to be responsible.

In addition to being used for deciding blame when failures occur, this information may also be made available to users to help them judge the consequences of their action or inaction. This information can be useful when a user is able to fulfill some of her obligations, but not all.

Chapter 3

Formal Model

We now present a highly abstract model of a user obligation system, which we call a metamodel. Any concrete model instantiates its features in a more specific way.

We model an obligation as a tuple $obl(s, a, O, t_s, t_e)$, in which t_s, t_e is the time interval during which subject s is obliged to take action a and O is a finite sequence of zero or more objects that are parameters to the action. A user obligation management system consists of the following components:

- $\mathcal{T}$: a countable set of time values, which we take to be the non-negative integers, with 0 being the system start time, and each other value indicating a time after that in some appropriate, unspecified time units.
- $\mathcal{S}$: a set of subjects from which system participants are drawn. We assume that each subject has a unique name, in the form of a string of characters, which identifies it.
- $\mathcal{O}$: a set of objects with $\mathcal{S} \subseteq \mathcal{O}$. We assume that each object has a unique name, in the form of a string of characters, which identifies it.
- $\mathcal{A}$: a finite set of actions that can be initiated by subjects. When an action is attempted, the system may accept or reject that action. When it is accepted, it causes a corresponding event.
- $\mathcal{B} = \mathcal{S} \times \mathcal{A} \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$: a set of obligations that can be introduced to the system. Given an obligation $b \in \mathcal{B}$, we use $b.s$ to refer to the subject that is obligated, $b.a$ for the action the subject is obligated to perform, $b.O$ for the finite sequence of zero or

more objects that are parameters to the action, and $b.t_s$ and $b.t_e$ to refer to the start and end, respectively, of the period in which the subject is obligated to perform the action.

- $\mathcal{ST} = \mathcal{T} \times \mathcal{FP}(\mathcal{S}) \times \mathcal{FP}(\mathcal{O}) \times \Sigma \times \mathcal{FP}(\mathcal{B})$: the set of system states. Here, we use $\mathcal{FP}(\mathcal{X}) = \{X \subset \mathcal{X} | X \textit{ is finite}\}$ to denote the set of finite subsets of the given set. We use $st = \langle t, S, O, \sigma, B \rangle$ to denote systems states, where t is the time in the system, S and O are the subjects and objects currently in the system, B is the set of pending obligations, and σ is a fully abstract representation of all other features of the system state. Σ, the domain of abstract states, is possibly infinite[1]. We use $st_{cur} = \langle t_{cur}, S_{cur}, O_{cur}, \sigma_{cur}, B_{cur} \rangle$ to denote the current state of the system.

- $\mathcal{E}$: a finite set of events. Events are derived from two sources. The first are *action events* which are caused by successful user actions. The second are uncontrollable events which occur outside the control of the system, but can affect system state, including causing obligations to be incurred. We designate the set of uncontrollable events $\mathcal{E}_U$. Recognizing that not all uncontrollable events might be able to occur at all times, we designate $\mathcal{E}_U(t)$ as the subset of $\mathcal{E}_U$ which can occur at time t.

  An event, e is a function which inputs the current state and a list of zero or more parameters and which outputs a new system state. Events cannot modify the system time. Action events always have a subject as a first parameter, this indicates the user who initiates the action. Beyond that, the parameters of an event can vary greatly from system to system. We only assume that each parameter is a finite string of characters. For example, an event might be an incoming sales order from a customer. In such a case, the parameters of such a request would likely include the identity of the customer and the items or services desired.

- $\mathcal{P}$: a set of access control policy rules. Each policy rule specifies an action that can be taken and under what circumstances it may be taken. Each policy rule p has the form

$$a(st, s, O) \leftarrow cond$$

  in which $a \in \mathcal{A}$ and *cond* (denoted by *p.cond*) is a predicate in $\mathcal{S} \times \mathcal{T} \times \Sigma \times \mathcal{O}^* \rightarrow$

[1] Σ would certainly be instantiated in any concrete model.

$\{true, false\}$, indicating that subject s is authorized to perform action a on objects O at time t with the system in state σ if $cond(s, t, \sigma, O)$ is true.

- $\mathcal{Q}$: a set of obligation policy rules. Each obligation policy rule specifies what obligations occur in response to a given event (either an action event or an uncontrollable event). Each obligation policy rule has the form

$$ae(st, Y) \wedge cond \rightarrow F_{obl}$$

Where ae is an event (a member of $\mathcal{A} \cup \mathcal{E}$), Y is its list of parameters, *cond* (denoted $q.cond$ is a predicate in $\mathcal{T} \times \Sigma \times (\mathcal{Y}^* \cup \mathcal{O}^*) \rightarrow \{true, false\}$ (where $\mathcal{Y}$ is the set of all finite strings), and F_{obl} is an *obligation function*, which takes the current state of the system σ, the current time, and the parameters Y as its input and returns a finite set $B \subseteq \mathcal{B}$ of obligations that must be taken on because of the action. Note that when the event is an action event, obligations in B may not necessarily be incurred by the same subject as performed the action.

An obligation $obl(s, a, O, t_s, t_e)$ may be in one of four states: *invalid*, *pending*, *fulfilled*, or *violated*. If it is the case that the t_e is already passed when it is assigned, then the obligation is *invalid*, as it, on its face, clearly can not be carried out. If an obligation has been assigned and its action has been carried out during the time window $[t_s, t_e]$, then it has been *fulfilled*. If it has been assigned, has not been *fulfilled*, and is not *invalid*, but t_e has passed, then it is *violated*. If an obligation is not *invalid* but has not yet become *fulfilled* or *violated*, then it is *pending*.

3.1 State Transitions

For simplicity, we assume that all actions scheduled for a given time can be finished in a single clock tick, and their effect will be reflected in the state of the next clock tick. Suppose the state of a system at time $t = i$ is st_i, and Alice takes an action at $t = i$. Then the state st_{i+1} at time $t = i + 1$ will be determined by the effect of Alice's action.

We also assume that all events occur in a single clock tick. If there is a happening which would take multiple clock ticks to complete, this should be modeled as a series of related events rather than as a single event. We assume that in order to determine the next

state of the system given the current one, first any uncontrollable events are applied to the state in the order they occured, and then allowed user actions are applied.

Suppose a finite set of actions are attempted at the same time, $t = i$, in state st_i. We denote such a set by $AP \subset \mathcal{S} \times \mathcal{A} \times \mathcal{O}^*$. ($AP$ stands for action plan.) Let us also assume that there is a finite list of uncontrollable events E_U occuring at the same time. We assume that these events, being outside events, occured in some order based on clock time and have their parameters known.

As such, we let there be a list of these events, $e_1, e_2, ..., e_{|E_U|}$ with a corresponding list of parameters $y_1, y_2, ..., y_{|E_U|}$. We apply the events of E_U to st_i in the order that they occured to derive a new state st'_i. That is, we define $st_i^0 = st_i$ and $st_i^j = e_j(st_i^{j-1}, y_j)$ for $j > 0$. Then we can define $st'_i = st_i^{|E_U|}$.

The order in which the elements of AP are executed is given by a fixed, arbitrary total order over $\mathcal{S} \times \mathcal{A} \times \mathcal{O}^*$. This means that two actions, if present in AP, will be executed in the same order regardless of other actions that may or may not be in the set. Let us assume $|AP| = n$ and $ap_0, ap_1, \ldots, ap_{n-1}$ enumerates AP in the order mentioned above. To determine which of the actions is permitted and what state the system is in after the permitted actions are performed, a sequence of states $\{\gamma_0, \ldots, \gamma_j, \ldots, \gamma_n\} \subset \Gamma = \mathcal{FP}(\mathcal{S}) \times \mathcal{FP}(\mathcal{O}) \times \Sigma \times \mathcal{FP}(\mathcal{B})$ is calculated. The first of these, $\gamma_0 \in \Gamma$, is obtained from st'_i by dropping $st'_i.t$, the time component. Each action in turn is then checked to see whether it is permitted in the state γ_j that is current at that point. This is done by determining whether there is a policy rule p the condition of which is satisfied by γ_j (that is, $p.cond(ap_j.s, st_i.t, \gamma_j.\sigma, ap_j.O) = true$). If it is permitted, one of the rules that has a satisfied condition is selected according to a fixed, deterministic procedure (which could involve user preferences), and the rule is applied to obtain obligations that are added to $\gamma_j.B$ to obtain $\gamma_{j+1}.B$. In addition, an obligation may be removed if ap_j satisfies it. The other components of γ_{j+1} are obtained by applying the action according to ap_j. The system state st_{i+1} is then obtained from γ_n by adding the time component $st_{i+1}.t = i + 1$. Given st_i and AP, we let the function $\mathsf{apply}(AP, E_U, st_i) = st_{i+1}$ in which st_{i+1} is obtained in this way. This defines the *transition* from st_i to st_{i+1} determined by AP and E_U. Given st_i, AP, and $ap \in AP$, we let $\mathsf{permitted}(ap, AP, E_U, st_i) = true$ just in case ap is permitted when it is attempted in the procedure described above. Otherwise, $\mathsf{permitted}(ap, AP, E_U, st_i) = false$.

Definition 1 *We say* $st \vdash_{AP} st'$ *is an* obligation-abiding transition*, if (1) there are no two tuples* (s_1, a_1, r_1) *and* (s_2, a_2, r_2) *in* AP *such that* $s_1 = s_2$ *and* $a_1 = a_2$*; and (2) for any* $(s, a, r) \in AP$*, there exists a pending obligation* $(s, a, [t_s, t_e])$ *in* $st.B$*.*

Definition 2 *An obligation-abiding transition is* valid *if no pending obligations in* st *become violated in* st'*.*

An *obligation-abiding* transition corresponds to the system evolution where subjects take actions (that is, contribute actions to AP) only to fulfill their obligations. A sequence of valid obligation-abiding transitions corresponds to the situation where subjects are diligent and always fulfill their obligations. An obligation-abiding transition is *valid* if no pending obligations in st_i become violated in st_{i+1}.

3.2 Accountability

Conceptually, a system's security policy divides system states into two disjoint sets: secure states and insecure states. The goal of security is to ensure that a system always stays in secure states and never transits into insecure states. Under the same principle, we study the question of how we should interpret a security policy which includes obligations, that is, what states are considered secure under policies with obligations.

One straightforward approach is to define secure states as those that have no obligations being violated. Such states are certainly desirable. However, due to the unenforceable nature of obligations, a user obligation management system can never guarantee that an obligation will be fulfilled. Instead, it seems more appropriate for such a system to ensure that all obligations *can* be fulfilled, in the sense that the obligated user has the necessary authorizations to perform the obligatory action. However, as we will see, this alone does not provide sufficiently clear guidance.

Certainly if the user obligation management system determines that it is impossible for a user to fulfill the obligation which would be incurred by his performing some requested action, then the reference monitor should deny that action. Likewise, if the system is certain that a user will have sufficient privileges to fulfill an obligation before its deadline, then the reference monitor should allow the requested action. But what is the appropriate thing for a user obligation management system to do if the ability of the user to perform the obligation depends on whether or not actions are taken to change his privileges?

Suppose there exists one sequence of actions or events which would cause the user to be unable to fulfill his obligations and another which would cause the user to be able to. One highly conservative response would be always to deny requests that would incur obligations that the obligated user might not be able to fulfill. However, in any system with a superuser, this would result in virtually every request being denied. In fact, so long as there existed any remedy which could take rights away from the user in any circumstance, all requests from that user for actions which carry obligations would be denied. One can imagine a scenario in which a CEO is denied the right to edit a file because it is theoretically possible that the board of directors could oust her from her position in the next five minutes.

Similarly, if one took the completely optimistic approach, the opposite scenario rears its head. So long as there exists some possible way that the user might be able to fulfill the incurred obligation, the requested action would be permitted, even if the events necessary for the user to have the required authorizations are highly unlikely. For example, Bob, a mailroom employee, could be allowed to carry out an action such as reserving the corporate jet even though he could only fulfill the associated obligation (using the corporate jet) if the board fired the CEO and hired Bob in his place within the next five minutes. As such, it is clear that neither strategy is acceptable.

In this thesis, we offer a set of assumptions which fall in between the two extremes of total optimism and total pessimism. Rather than requiring that it be impossible for obligations to be violated, instead we assume that it is possible that obligations go unfulfilled, but when they do, we would like to clearly identify whose fault it is. Obviously, an obligation can go unfulfilled because a subject simply fails to take the required action before the deadline, even if he has sufficient privileges and resources. It is desirable that this is the only reason that an obligation will go unfulfilled.

With this goal in mind, we propose a new property of system states which we call *accountability*. We name it this way because in a system whose state is accountable if there is a single obligation failure, then we can know that it is the fault of the user to whom that obligation was assigned. However, this is more specific than the general idea that the users of the system can be held accountable for their actions, in the normal sense. As such, we will provide a formal definition. It should be assumed for the remainder of this thesis that any time we discuss accountability, we are using it in the sense that we are defining.

Intuitively, if it is the case that all users have sufficient privileges and resource to

carry out their obligations so long as every other user carries out his or her obligation, then a system is said to be in an *accountable state*, because we can know that whoever first fails to carry out an obligation is responsible for the violation and anything which results from it.

Note that when determining whether a state is accountable, we only consider those actions that are required to be taken by obligations. Although a user may actively take some actions which may interfere with another user's obligations, such actions can be controlled by the reference monitor. Once the system determines whether the resulting state will be accountable or not, it can take appropriate actions. For example, it may either prevent the user from taking the action, or it may discharge or change the interfered obligations so that the resulting state is still accountable. In this section, we focus on the definition and checking of accountable states. We will briefly discuss the handling of actions that may lead a system into an unaccountable state.

Before we give a formal definition of accountable states, it is necessary to discuss in more detail in what situation it is that a user is deemed as failing to fulfill her obligation. Intuitively, when an obligation $(s, a, [t_s, t_e])$ is assigned, we can view it as a contract between a subject s and a system. From the subject's perspective, it has promised to take action a during the given time window. On the other hand, from the system's perspective, it also implicitly promises that if everybody else fulfills their obligations, then s should have the needed privileges and resources to take action a. Depending on the interpretation of obligations, we may have different definitions of accountability.

Here we consider two types of interpretations. In the first type, if everybody else fulfills their obligations, then a system guarantees that Alice can take action a at any time point during $[t_s, t_e]$. This is a very strong promise from the system, which means that the condition of the rule for action a is always true for Alice during the obligation's time window.

In the second type, a system only promises that, if everybody else fulfills their obligations, then Alice can at least take action a at the end point t_e of the obligation's time window. Note that it does not mean that Alice has to take the action at t_e; it only means that in the worst case Alice will still be able to fulfill the obligation at t_e. Clearly, this type of promise is weaker than the first one, since it only requires that the condition of the rule for action a is true at t_e instead of during the whole time window of the obligation.

Generally speaking, the first type of accountability is suitable for systems in which users may have additional restrictions of which the system is not aware. If there are additional constraints on when a user is available to fulfill her obligations, then the system should ensure that the full time is available to the user so that whenever the user has the opportunity to fulfill one of her obligations, she has the necessary authorizations to do so. For instance, a system at a company with flexible work hours would probably prefer strong accountability since it would not know when employees would be available to fulfill their obligations. By contrast, in a more all-encompassing system, the weaker accountability is sufficient: since the system is aware of all scheduling constraints, whenever the user chooses to attempt to fulfill the obligation, either she will be able to fulfill it at that time, or the system will have ensured that she will again be available at a later time prior to the deadline. Hence, a system at a military base, where all users can be expected to be available precisely when the policies say that they should, might be able to use weak accountability, and derive benefit from the fact that it is a weaker requirement and therefore easier to ensure.

There is a third possible type of accountability, in which the system ensures only that there exists some time within the frame when the user will be able to fulfill his obligation. This type, however, is not likely to be generally suitable for ordinary users since it would require that the user discover that time before it passes. It may be suitable for systems with automated agents which could regularly poll the system to see if the obligation can be fulfilled. Although this type of accountability sounds straightforward, it actually turns out to be quite complicated to formalize. As such, it is not covered in this dissertation.

Next, we formally define accountable states for each of the first two interpretations. Recall from Definition 1 that a transition is valid and obligation-adbiding if all the actions in its action plan are from existing obligations and no obligations are violated.

Definition 3 *Let st be a system state with time $st.t = t$ and pending obligations $st.B = \{b_1, \ldots, b_n\}$. We say st is a* type-1 undesirable state *if there exists $B' \subseteq \{b \in st.B | b.t_s \leq t \leq b.t_e\}$ and, letting $AP' = \{(b.s, b.a, b.O) | b \in B'\}$, there exists $ap \in AP'$ and $E_U \in \mathcal{E}_U(t)$ such that $\mathsf{permitted}(ap, AP', E_U, st) = \{false\}$.*

A state is type-1 undesirable if a subject cannot fulfill an obligation although the current time is within the time window of the obligation.

Definition 4 *A state st is* strongly accountable *if there exists no sequence of valid obligation-*

abiding transitions that lead st to a type-1 undesirable state.

This definition of accountability corresponds to the first interpretation of obligations. For the second interpretation, we have the following definition.

Definition 5 *Let st be a system state with time* $st.t = t$ *and pending obligations* $st.B = \{b_1, \ldots, b_n\}$. *We say st is a* type-1 undesirable state *if there exists* $B' \subseteq \{b \in st.B | b.t_s \leq t \leq b.t_e\}$ *and, letting* $AP' = \{(b.s, b.a, b.O) | b \in B'\}$, *there exists* $ap = (b.s, b.a, b.O) \in AP'$ *and* $E_U \in \mathcal{E}_U(t)$ *such that* $\mathsf{permitted}(ap, AP', st) = \{false\}$ *and* $t = b.t_e$.

A state is type-2 undesirable if the current time is the deadline of a pending obligation, which however cannot be fulfilled at present.

Definition 6 *A state st is* weakly accountable *if there exists no sequence of valid obligation-abiding transitions that lead st to a type-2 undesirable state.*

Obviously, if a state is strongly accountable, it is also weakly accountable. Let us consider the following scenarios. Suppose in state *st* at time 0 Alice does not have the privilege to read file f, but she has an obligation to read f between time 10 and time 20. Meanwhile, Bob, whose is the owner of f, has an obligation to grant the read privilege of f to Alice between time 5 and time 15. According to our definition, *st* is not strongly accountable, as it is possible that Alice decides to fulfill her obligation at time 12, but she cannot do so because she lacks the read privilege. In this case Bob is not to blame, since Bob can decide to fulfill his obligation at time 14, for example. In other words, *st* may possibly transit into a future state where a subject cannot fulfill his obligation, but this is not due to any subject's negligence.

On the other hand, *st* is weakly accountable. This is because Alice can always fulfill her obligation at time 20. If she still does not have the privilege to read f at time 20, it must be due to Bob's negligence.

Chapter 4

Accountability Checking

In this chapter, we discuss questions related to evaluating whether or not a system is accountable. We start with a completely abstract situation and then later examine more specific cases. In the first several sections of this chapter, we assume that the set of uncontrolable events, $\mathcal{E}_U$ is the empty set. Dealing with events further complicates what is already a complex problem, and, as such, we save the discussion of accountability checking with events for its own section, section 4.3.

4.1 The Accountability Problem

In this section, we study the *accountability problem*, that is, given a state in a system, determining whether it is accountable. This problem is naturally faced by a reference monitor when a subject makes a request to take an action which, if allowed, would result in the assignment of obligations.

Note that since the constructs of our metamodel are very abstract, it can accommodate systems with arbitrarily complex internal structures. It is not hard to see that, purely based on the metamodel without any constraints on a system's properties, the accountability problem can easily be undecidable. In the following, we show a reduction of the halting problem to the accountability problem.

Given a Turing machine T, we can construct a system which emulates that Turing machine by specifying that the system state, σ, includes a potentially infinite tape, a position on that tape, a current machine state, and a boolean variable which describes whether or

not the system has halted. Then we define two actions. The first is "Advance", which changes the state and the tape in accordance with the state transition rules of T and, if T halts, sets the halt variable to true. The second action is "Fail", which has no effect. For purposes of simplicity, we define a single subject, s. To make the Turing machine operate, we define an initial obligation of $(s, Advance, [1, 1])$.

Then we define the policy rules for Advance so that advancing causes an obligation to advance the state again or, if the halt flag has been set, an obligation to fail. Explicitly, the policy rules are:

- $Advance(s, \emptyset) \leftarrow true$
- $Advance(s, \emptyset) \rightarrow f(\sigma, t, s) = \{(s, Advance, t + 1, t + 2)\}$
- $Advance(s, \emptyset) \leftarrow halt$
- $Advance(s, \emptyset) \rightarrow f(\sigma, t, s) = \{(s, Fail, t + 1, t + 2)\}$
- $Fail(s, \emptyset) \leftarrow false$
- $Fail(s, \emptyset) \rightarrow f(\sigma, t, s) = \emptyset$

In the above, t denotes the current time of a system. We have previously required that there be a deterministic way to chose which rule should be applied if multiple rules are satisfied.

In this particular case, if the first and second rules are both satisfied then the second rule should always be used. As such, if the machine ever halts, there will be an obligation which cannot be fulfilled. But if it never halts, there will not be one. Therefore, the question of whether or not the state is unaccountable is equivalent to the question of whether or not the Turing machine will halt. As such, in the worst case, the accountability problem is undecidable, when only considering the constructs of the metamodel without any constraints.

To describe such a reduction is, of course, by no means to say that the accountability problem is undecidable in all obligations systems. In specific systems, determining accountability may often be quite easy. In particular, we are interested in identifying the properties of obligation systems which allow us to efficiently solve the accountability problem.

Let us consider obligation systems that satisfy the following conditions:

- No cascading obligations. In the metamodel, the action to fulfill an obligation may also incur further obligations. If a system does not have such cascading obligations, then each obligation only involves actions whose policies do not carry obligations.

- Monotonicity. From a given state, if the condition on a policy is true for a subject, it will remain true in all future states. In other words, the set of rules that a subject can satisfy does not decrease during state transitions. As such, the set of rules is monotonic relative to time.

- Commutative and time-independent actions. For any two actions, a_1 and a_2 if the conditions of the policy rules of both a_1 and a_2 are met, then taking action a_1 followed by a_2 has the same effect on the system state as taking action a_2 followed by a_1. And, for any action a, $a((t_1, S, O, \sigma, B), s, O') = a((t_2, S, O, \sigma, B), s, O')$ for all $t_1, t_2 \in T$. Also, the conditions associated with actions should be time independent. That is, for any given policy rule $p \in \mathcal{P}$ which has condition *cond*, $\forall t_1, t_2 \in \mathcal{T}, \sigma \in \Sigma, s \in \mathcal{S}, O \subset \mathcal{O}$, $cond(s, t_1, \sigma, O) = cond(s, t_2, \sigma, O)$.

These properties might seem a little draconian. But as we will show later, if we remove any one of them, without considering other specifics of a system, the accountability problem is intractable. Again, this does not mean that particular systems which do not have these properties cannot be efficiently solved, since any particular system would have additional properties which we do not assume here. In fact, in section 4.2 we will present a class of systems which do not conform to all of these assumptions, but do have an efficient algorithm for the accountability problem. With that said, any time we have a system where the above three properties hold, we can efficiently solve the accountability problem, without looking at other specifics of that system.

Theorem 1 *Given a system that satisfies the above three properties, the problem to check whether a given state is weakly accountable is tractable.*

Proof 1 *In a monotonic system, once an obligatory action becomes enabled, it remains so in later states. Thus, weak accountability in this context is equivalent to requiring that each obligatory action is enabled at the end of its time window. As such, it should be the case that the "worst-case" schedule is one in which all obligations are scheduled for their last*

possible time slots. To formalize this, we first define the set of possible actions in a given state.

Definition 7 *Let X be the set of all tuples of the form (s, a, O) where $s \in \mathcal{S}$, $a \in \mathcal{A}$, $O \subset \mathcal{O}$. We define L_X to be the lattice $(2^X, \bigcup, \bigcap)$. This lattice is ordered via subset inclusion.*

Next we define a function $L : \Sigma \rightarrow L_X$ such that for any $\sigma \in \Sigma$ $L(\sigma)$ is the set of all $x = (s, a, O) \in X$ such that there exists $p \in \mathcal{P}$ such that p governs action a and $cond(s, t_0, \sigma, O)$ is true, where t_0 is a dummy time value. Because of the time independence of conditions imposed as part of our commutativity property, we know that the value of cond is the same regardless of the value chosen for t_0.

Next we note that because of the fact that there are no cascading obligations, there must be some finite set of currently pending obligations B which is a subset of $\mathcal{B}$. However, for purposes of simplicity, in this proof we are going to define an alternate set of possible obligations, $\mathcal{B}^*$. We do this to simplify issues related to the situation in which two actions occur during the same time slice. The facilitate this, we are going to define a new set of times, $\mathcal{T}^*$. $\mathcal{T}$ is the set of all possible system times. However, it would be convenient to have a set of times which properly reflect the ordering of actions within a single time slot. We have previously assumed that there is some ordering of X such that we can know in what order two actions will occur if they are both attempted in the same time slot. Let us use $\leq$ over X to refer to this ordering. We therefore define a new set of time values, $\mathcal{T}^* = \mathcal{T} \times X$. $\mathcal{T}^*$ is ordered as follows:

Definition 8 *For any two $t_1^* = (t_1, x_1)$,$t_2^* = (t_2, x_2)$ $t_1^* \leq t_2^*$ if and only if $t_1 < t_2$ or both $t_1 = t_2$ and $x_1 \leq x_2$.*

As we have taken $\mathcal{T}$ to be the natural numbers, we may think of $\mathcal{T}^*$ as being a version of $\mathcal{T}$ where we have added a value after the decimal place which lets us distinguish times with a finer resolution.

By previous definition, $\mathcal{B} = \mathcal{S} \times \mathcal{A} \times 2^{\mathcal{O}} \times \mathcal{T} \times \mathcal{T}$. Correspondingly, we define $\mathcal{B}^*$ to be the subset of $\mathcal{S} \times \mathcal{A} \times 2^{\mathcal{O}} \times \mathcal{T}^* \times \mathcal{T}^*$ such that $\forall b^* = (s, a, O, t_s^*, t_e^*) \in \mathcal{B}^*$ such that $t_s^* = (t_s, (s, a, O))$ and $t_e^* = (t_e, (s, a, O))$ for some $t_s, t_e \in \mathcal{T}$. That is, we define $\mathcal{B}^*$ to be the set of obligations whose starting and ending time are modified to reflect the ordering of actions within time slots. As such, there is a bijective mapping between $\mathcal{B}$ and $\mathcal{B}^*$ which can

be formed by simply mapping t_s and t_e to $(t_s, (s, a, O))$ and $(t_e, (s, a, O))$. For the remainder of this proof, we will treat $\mathcal{B}$ and $\mathcal{B}^*$ as equivalent. Due to the ordering constraints, if an obligation $b = (s, a, O, t_e, t_s)$ occurs at time $t \in \mathcal{T}$, then we will treat it as occurring at time $(t, (s, a, O)) \in \mathcal{T}^*$.

Hence, we define B^* as being the set in $\mathcal{B}^*$ equivalent to B. We define a lattice L_{B^*} to be the lattice $(2^{B^*}, \bigcup, \bigcap)$. Intuitively, our lattice, L_{B^*}, is going to represent which obligations have been fulfilled. As such, over time, we can only move up the lattice, since previously fulfilled obligations cannot become "unfulfilled."

Given an initial permission state, σ_0 and a set of obligations which have been carried out, l_{B^*}, we can exactly determine the state which will result, which we call $\sigma(\sigma_0, B^*)$. In order to define $\sigma(\sigma_0, l_{B^*})$, we first let there be some ordering $b_1, b_2, ..., b_n$ of obligations in l_{B^*} such that if we define $\sigma_i = a_i(\sigma_{i-1}, s_i, O_i)$ where $b_i = (a_i, s_i, O_1)$ then $\forall 1 \leq i \leq n, \exists p \in \mathcal{P}$ with condition cond and action a_i such that $cond(s_i, t_0, \sigma_{i-1}, O_i)$. Again, t_0 serves as a dummy time variable due to the time-independence property of actions. That is, we let there be an ordering such that all the actions can happen in accordance with the security policy. If such an order exists, then we define $\sigma(\sigma_0, l_{B^*}) = \sigma_n$. If no such order exists, then we define $\sigma(\sigma_0, l_{B^*}) = \sigma_0$. Please note that the second case is not relevant to real situations which arise in obligations systems since, but is included here for purposes of completeness. We should further note that if there exists more than one possible order, then due to the commutativity property, the resultant value of σ_n will be the same under all orders.

Now we may combine the two mappings. We know that given any initial permission state, σ_0, there is a mapping from L_{B^*} to L_X. We define $G_{\sigma_0}(l_{B^*}) = L(\sigma(\sigma_0, l_{B^*}))$.

Next we wish to examine how things vary over time. We define a function, $F(t)$ which maps time values from T^* to subsets of B, in such a way that the transitions could be carried out in a real system. That is, for any time t, there exists an ordering $b_1, b_2, ..., b_n$ of the members of $F(t)$ as described previously. $F(t)$ is meant to represent the fulfillment of obligations over time. As such, we restrict $F(t)$ to be monotonic with respect to time, since one cannot "unfulfill" obligations. That is, $\forall t_1, t_2 \in T^*, t_1 \leq t_2 \rightarrow F(t_1) \subseteq F(t_2)$.

Next we note that due to the property of monotonicity, for any σ_0, $G_{\sigma_0}(F(t))$ must also be monotonic relative to time. That is, $\forall t_1, t_2 \in T^*, t_1 \leq t_2 \rightarrow G_{\sigma_0}(F(t_1)) \subseteq G_{\sigma_0}(F(t_2))$.

Due to their monotonicity, it follows that for any possible range of t, both $F(t)$

and $G_{\sigma_0}(F(t))$ achieve their minimal value when t is minimal. Further, the minimal $F(t)$ must therefore always yield a minimal $G_{\sigma_0}(F(t))$. As such, we get the following theorem:

Theorem 2 *For any $l_B, l'_B \in L_B$, if $l_B \subseteq l'_B$ then $G(l_B) \subseteq G(l'_B)$.*

From this, we can know that for any particular (a, s, O) combination, such as we might find in an obligation, that if we wish to find out whether or not there is a schedule for which its condition is not satisfied at some time t, then we should examine the schedules which lead to minimal values for $F(t)$, that is, the ones in which the fewest obligations have been completed before time t.

Next we wish to use the above theorem to show that if the schedule of obligations in which every obligation is done last does not result in any obligation failures, then no schedule will.

Let us therefore consider that we have some set of obligations, B, and that we have ordered them of by their end times. Obligations which have the same end time will be ordered according to when they would execute if they were both attempted at the same time. [1]

Given the first such obligation, $b = (a, s, O, [t_s, t_e])$, when we examine the state at its end time, t_e, clearly the minimal $F(t_e)$ can be obtained from a schedule in which all other obligations happen after it, since in that case $F(t_e) = \emptyset$. If this obligation's condition is met under such a schedule, then it follows that the condition is met under any schedule. This is true because the condition being met implies that $(a, s, O) \in G_{\sigma_0}(\emptyset)$ for the particular σ_0. As such, (a, s, O) must be in $G_{\sigma_0}(l_B)$ for all $l_B \in L_B$ since $\emptyset \subseteq l_B \forall l_B \in L_B$ and this implies that $G_{\sigma_0}(\emptyset) \subseteq l_B \forall l_B \in L_B$ If the obligation's condition is not met, then the system is clearly unaccountable because there exists a valid schedule in which at least one obligation failure occurs.

Next, we consider the remaining obligations in sequence. For the nth obligation, $b = (a, s, O, [t_s, t_e])$ the minimal $F(t_e)$ is the set of obligations which are before it. Obligations whose deadlines fall before a given obligation's deadline must occur before it in any schedule in which all obligations occur. We can also assume that either in examining previous obligations we have uncovered that the system is unaccountable, in which case we do not

[1] In defining the system, we have required that there be a fixed order of execution in which actions requested during the same time slice are to be carried out.

need to consider this case, or that the conditions of all previous obligations were met in the schedule in which they were all scheduled to occur at their deadline. This implies that there does exist some schedule of the obligations which come before b which meets the condition outlined.

Obviously, if the condition is not satisfied by this schedule, then the system is unaccountable. If, however, the condition is satisfied in this schedule, then it will be satisfied in any schedule. We know this because the condition being met implies that $(a, s, O) \in G_{\sigma_0}(F(t_e))$ *for the particular* σ_0 *and our minimal* $F(t_e)$*. Because our* $F(t_e)$ *is minimal, our* $G_{\sigma_0}(F(t_e))$ *must also be minimal, which is to say that any other reachable member of* L_X *is a superset of* $G_{\sigma_0}(F(t_e))$ *and therefore also contains* (a, s, O)*.*

Therefore, if we consider the given schedule and check whether or not each obligation's condition will be satisfied at their end time in such a schedule, then we can know whether or not the state is accountable.

Assuming it takes a constant time to check whether a condition is satisfied in a state, the complexity of the above algorithm is $O(nm)$*, where* n *is the number of pending obligations in a state, and* m *is the number of action rules in the policy.*

We have a similar result for the checking of strong accountability.

Theorem 3 *Given a system that satisfies the above three properties, the problem to check whether a given state is strongly accountable is tractable.*

Proof 2 *We reuse the techniques and theorems from the weak accountability proof. In this case, however, instead of examining whether or not an obligation's condition is true at its end point, we examine whether or not it is true at its start point. By the property of monotonicity, if it is true at its start point, then it must remain true throughout. And if it is not true at its start point, then clearly there exists a time at which the obligation cannot be carried out: its start point.*

To test strong accountability, we again schedule each obligation at its end point. And then one by one, we examine obligations to see if their condition holds at their start point, t_s *in such a schedule. This will again yield a minimal member of* L_{B^*} *for each point in time, since all obligations whose end point falls before the given* t_s *must occur, and no others are required. By the same logic as before, if the condition is met at that point in this*

schedule, then it is met at that point in all schedules. And if it is not met, then we are not strongly accountable.

Assuming that the checking of a condition against a given state requires constant time, the complexity of this algorithm is $O(nm)$, where n is the number of pending obligations in a state, and m is the number of action rules in the policy.

4.1.1 Partial Assumptions

If we remove one of the above three properties, then the accountability problem becomes intractable, when only given the constructs of the metamodel.

Theorem 4 *Given a system which only satisfies two of above three properties, that is, the no cascading obligation, the monotonicity and commutative action properties, the problem of determining whether a state of the system is strongly/weakly accountable is intractable.*

There are three possible ways to choose two out of the three properties. In the following, we show that in one of the cases, the problem becomes undecidable and in the other two, it is Co-NP complete.

4.1.2 Without the Property of No Cascading Obligations

In the circumstance that cascading obligations are allowed, even when the other two properties hold, the problem is still undecidable. Our construction for this is actually the same construction we used to prove undecidability earlier. In that construction, both of the other two properties hold. As one of the actions has no effect on the state, clearly the actions are commutative. Also, the only condition which changes is the condition on the second policy rule. Because the halt variable is initial set to false, the condition starts out false and can only become true. As such, we can know that these two assumptions alone are not sufficient to make the problem decidable.

4.1.3 Without the Property of Monotonicity

First, we establish that the problem is in Co-NP. If a system is unaccountable, that indicates that there must exist a schedule of obligations which causes some condition to go unsatisfied. Such a schedule can be uniquely described by listing obligations and the

times when they occur. Whether or not a given schedule will result in an obligation failure can be checked in time polynomial in the number of obligations by simulating the system state. Because we have the property of no cascading obligations, we can know that the only obligations which need to be scheduled are those which currently exist, and hence the schedule will be linear in the size of the problem. As such, the schedule serves as a short certificate, and therefore the problem must be in Co-NP.

Next we show that for systems in which monotonicity is not present, the set covering problem can be reduced to the unaccountability problem, and hence the unaccountability problem is Co-NP Hard.

In set covering problem, there is a set, S, and a set of n subsets of S, $\{S_1, \ldots, S_n\}$. There is also a parameter, k, and the question is whether or not there is a set of k of the subsets such that the union of those sets is equal to S. That is, is every member of S "covered" by one of the k subsets.

In this case we are not going to have cascading options. So we cannot have a system in which the obligations force us to make choices between different options.

As such, our state, σ is represented by a set S' and an integer m. We have a single subject, s. We have $n + 1$ actions. For $i = 1, \ldots, n$ we define an action $remove_i$ which removes S_i from S' and increments the value of m by one. We also define an action *check* which has no effect.

Next we define the policies for the actions. For $i = 1, \ldots, n$ we define policies $remove_i(s) \leftarrow true : f(\sigma, t, s) = \emptyset$ and $check(s) \leftarrow \neg(m = k \wedge S' = \emptyset) : f(\sigma, t, s) = \emptyset$.

Our initial state is the state such that $m = 0$, $S' = S$ and we have a set of obligations $\{(s, remove_i, 0, 2n) | i = 0, \ldots, n\} \bigcup \{s, check, 0, 2n\}$. All of the subset actions can be fulfilled. But only those which are fulfilled before the check obligation matter to the check of accountability. So, the question of accountability is the question of whether or not it's possible to schedule k of the subset obligations before the check obligation which together cover S. If it is possible, then the initial state is unaccountable. If it is impossible, then the initial state is accountable, since that means that there is no schedule which will result in an obligation failure. As such, we have reduced Set Cover to unaccountability.

Again we finish by verifying that our other two assumptions hold in our system. Clearly, there are no cascading obligations as no policy contains any new obligations. And it is the case that all actions are commutative since for any two actions, $remove_i$ and $remove_j$,

the result of apply them is that S' takes on the value of S'
$(S_i \bigcup S_j)$ and m takes on the value $m + 2$.

4.1.4 Without the Property of Commutative Actions

Accountability in systems with monotonicity and no cascading obligation, but without the property of commutative actions is in Co-NP for the same reasons stated at the beginning of the last section. The same schedule described there also serves as a short certificate in this case.

Now we must prove the accountability problem under these assumptions is Co-NP hard. In order to do this, we use satisfiability in our reduction. We assume the existence of a set of n boolean variables $\{x_1, \ldots, x_n\}$ and a boolean expression $X(x_1, \ldots, x_n)$.

We define our state to have a single user s. And we define our system state, σ as a set of boolean variables, $\{x_1, \ldots, x_n\}$ and a counter, m. We define $2n+1$ actions. The first $2n$ are used for setting the variables to either 0 or 1. For $i = 1, \ldots, n$, we define actions set_i^0 which sets x_i to be 0 and increments m by 1 and set_i^1 which sets x_i to 1 and increments m by 1. Our last action is *check*, and it has no effect.

We define, for $i = 1, \ldots, n$, the following policy rules:

$$set_i^0(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$set_i^1(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$check(s) \leftarrow m \geq 2n \wedge \neg X(x_1, \ldots, x_n) : f(\sigma, t, s) = \emptyset$$

Lastly, we define our initial state. In our initial state, $t = 0$, $x_i = 0$ for $i = 1, \ldots, n$, and we have a set of $2n$ initial obligations, $\{(s, set_i^0, 3i, 3i+2) | i = 1, \ldots, n\} \bigcup \{(s, set_i^1, 3i, 3i+2) | i = 1, \ldots, n\} \bigcup \{(s, check, 3n, 3n+5)\}$. The question of whether or not this initial state is unaccountable is equivalent to the question of whether or not X is satisfiable. In short, the obligations are going to require that each x_i be set to either 1 or 0. In fact, at some point it will be set to both 0 and 1, but only whichever assignment appears last will be relevant to the condition of *check*. Because the final obligation has that X be unsatisfied as its condition, the initial state will only be accountable if there is no assignment of values which will cause X to be satisfied. As such, if the initial state is unaccountable, then X is satisfiable.

With our construction complete, we once more take a moment to note that it is consistent with our other two assumptions. The system is monotonic because there is no valid sequence of obligation actions going forward from our particular initial state which can cause any condition to go from true to false. It's also clearly the case that there are no cascading obligations as no policy carries any obligation.

4.2 A Concrete Model

As we saw in the previous section, in the abstract model, determining the accountability of a system can be done efficiently provided several restrictions are placed on the system. In particular, one requirement was that actions perform state transitions that cause the set of enabled actions in the system to either increase (with respect to $\subset$) or stay the same. In the abstract model, this could not be relaxed without losing tractability. However in practice the restriction is unlikely to be satisfactory, as it means that once a subject is able to perform a given action, they will always be able to perform that action.

By contrast with our meta-model, in practice, permission states are structured objects. It turns out that entirely realistic assumptions about that structure enable us to remove the assumption that actions must increase while preserving tractability.

In this section we present a concrete model based on the HRU access matrix model. Specifically, Σ, the set of abstract states, is instantiated to be $\mathcal{M} = 2^{\mathcal{S}\times\mathcal{O}\times\mathcal{R}}$, the set of permission sets, in which $\mathcal{R}$ is a set of access rights subjects can have on objects. We denote permission sets by M and individual permissions by $m = (s, o, r)$. Each permission is a triple consisting of a subject, an object, and an access right, and signifies that the subject has the right on the object.

Actions are also modified so as to operate on permission sets. Each action $a \in \mathcal{A}$ is now assumed to perform a finite sequence of operations that each either add or remove a single permission from the permission set ($grant(m)$ and $revoke(m)$). Clearly, a subject or an object with no associated permissions has no effect on the system, so we assume that in every state st, an object or a subject exists in $st.O$ and/or $st.S$ if and only if it occurs in some permission in the permission set $st.M$.

As we show below, the following restrictions on the model make the problem of determining whether a state is accountable tractable.

1. Policy rule conditions consist of a Boolean combination of permission tests ($m \in M_{cur}$ or $m \notin M_{cur}$) expressed in conjunctive normal form.

2. Actions are partitioned into two sets—the first consists of actions whose policy rules can impose obligations and the second consists of actions that can occur in those obligations. This means that one cannot become obligated to take actions in the first set, but performing such actions voluntarily can incur obligations to perform actions in the second set. Performing these latter actions does not incur any obligations.

4.2.1 Algorithm for dealing with the simplified concrete model

Given a current state $st_{cur} = \langle t_{cur}, M_{cur}, B_{cur} \rangle$ that is known to be accountable, we can use the procedure below to determine whether or not adding a new obligation b leaves the system in an accountable state. Given a set of obligations B that need to be added, we can use the procedure by considering the elements of the set one at a time, adding each obligation in turn to B_{cur} unless it would leave the system in an unaccountable state, in which case we stop and return the result that B cannot be added while preserving accountability.

We are given an obligation $b = \langle b.s, b.a, b.t_s, b.t_e \rangle$. Let the policy rule that governs $a = b.a$ be $a \leftarrow cond(s, t, M)$. Under the restrictions identified above, $cond(s, t, M)$ is a Boolean combinations of permission membership tests expressed in conjunctive normal form (that is, a conjunction of disjunctions). The following steps are used to check each permission membership test in turn to see whether it is guaranteed. The condition is guaranteed to be true if and only if all of its conjuncts is true. Since any one conjunct (/ie, any one disjunction) is guaranteed to be true only if it is true in all possible schedules of obligations, we can know that we can evaluate the truth of each conjunction without regard to any other. That is, we are seeking to answer the question of whether or not there exists a schedule which makes any one of the conjuncts false.

We examine each of the disjunctions individually by initially examining all of the tests in a given disjunction individually. If a test is guaranteed to be true, then we know that the disjunction is guaranteed to be true, and we can move on to the next disjunction. If we find that a test is guaranteed to be false, then we know that we can disregard it. If however, we find that we do not know, we set that test aside and later do a more complex

check.

This more complex check is necessary because it is possible that although it may be the case that some pair of rights r_1 and r_2 may each, individually, not be guaranteed to exist in all possible schedules, $r_1 \vee r_2$ could still be guaranteed. For instance, if there were an obligation which both revoked r_1 and granted r_2 and another obligation which both granted r_1 and revoked r_2 such that they had overlapping time intervals, then neither r_1 nor r_2 would be guaranteed, but $r_1 \vee r_2$ would be.

To test an individual right, all we are doing is checking existing obligations to see what state the last obligation to touch that permisison has left it in. If the last obligation is unique and it has granted or revoked the right we need, our answer is clear. However, it may not be certain which action will be carried out last since obligations with conflicting results may overlap. For instance, if there are two overlapping obligations, one of which grants a given right and the other of which revokes it, then one cannot know beforehand whether or not the right will exist after the obligations have been fulfilled. If such a situation arises, then we apply our more complex analysis to see if the condition is guaranteed to be satisfied even though the individual tests are not guaranteed.

Broad Algorithm

In the following, we assume a function $CheckDisjunction(O, M', t, \sigma, t_0)$, which returns true if there exists a schedule of obligations in O which causes all of the permissions in M' to not exist at time t when starting from state σ at time t_0. The algorithm to compute this function is discussed after the general algorithm.

1. Check condition. We repeat the following for each disjunction in the condition.

   (a) Check if the disjunction contains any two tests, m and $\neg m$ for any permission m. If it does, skip to the next disjunction.

   (b) Define a set of rights, M', and initialize to $\emptyset$.

   (c) We do the following for each test which is part of the disjunction. We assume the test is positive (that is, let it take the form $((s, o, r) \in M)$. If in fact the test is negative $((s, o, r) \notin M)$, then the procedure is obtained from the following by reversing the roles of "grant" and "revoke" and negating permission membership tests. We refer to the permission being tested as m.

i. If there is an overlapping revoke action, that is, some $br \in B_{cur}$ has $br.a = revoke(b.s, o, r)$ and the intervals $[b.t_s, b.t_e]$ and $[br.t_s, br.t_e]$ intersect, than m cannot be guaranteed, and m should be added to M'.

ii. Otherwise, if the privilege exists in the current state, that is, $(b.s, o, r) \in M_{cur}$ then

   A. If there is a prior revoke action, that is, some $br \in B_{cur}$ with $br.a = revoke(b.s, o, r)$ has $br.t_e < \mathbf{b.t_s}$, then pick such a $br \in B_{cur}$ so as to maximize $br.t_e$ (subject to $br.t_e < \mathbf{b.t_s}$). The test can be guaranteed only if there exists an obligation that someone grant the permission (again) after br but before b, that is, only if some $bg \in B_{cur}$ has $bg.a = grant(b.s, o, r)$, $br.t_e < bg.t_s$ and $bg.t_e < \mathbf{b.t_s}$. If the test can be guaranteed, then proceed to the next disjunction. If it cannot be guaranteed, then add m to M'.

   B. Otherwise, the test can be guaranteed, and we proceed to the next disjunction.

iii. Otherwise, if the privilege does not exist in the current state then

   A. If there is some grant obligation for the tested permission $bg \in B_{cur}$ that ends before b starts, then pick some bg so as to maximize $bg.t_s$ while satisfying $bg.t_e < b.t_s$. The test can be guaranteed only if no revoke obligation for the tested permission $br \in B_{cur}$ that overlaps with the interval $[bg.t_s, b.t_e]$. If the test can be guaranteed, then we proceed to the next disjunction. If the test cannot be guaranteed, then we add m to M'.

   B. Otherwise, the test cannot be guaranteed, and we add m to M'.

iv. We evaluate whether or not there exists any point t' between $b.t_s$ and $b.t_e$ at which the condition is false. If such a t' exists, it follows that there exists a such t' which is equal to either the start or end of b or the start or end point of some other obligation, such that that start or end point is in between $b.t_s$ and $b.t_e$. Because this is true we limit our checking to such points. As such, we define a set T' to be all such start and end points (including $b.t_s$ and $b.t_e$).

We evaluate $CheckDisjunction(O, M', t', \sigma, t_0)$ for all $t' \in T'$ where O is the set of all pending obligations other than b, σ is the current state, and t_0 is the current time. If the result is $true$ for all t' then we pass the test and move to the next disjunction. If the result is $false$ for any t' then we are not accountable and we terminate.

(d) If there are no disjunctions remaining, then we are accountable and we terminate.

2. Check effect of b on overlapping and later obligations. The obligation b likely either grants or revokes some right. Obligations which depend on the presence or absence of this right need to be considered. To check them, we repeat step 1 of this algorithm for each obligation whose starting time is after b's starting time.

Helper Function

Next we must describe how to construct the $CheckDisjunction(O, M', t', \sigma, t_0)$ function. First, we remove any irrelevant obligations. We begin by removing any obligations from O which neither grant nor revoke any right in M'. Then we remove obligations which are outside of our time frame. $O = O \backslash \{o \in O | o.t_e < t_0 \vee o.t_s > t'\}$. Then we define our base cases. If $M' = \emptyset$ then we return $false$. If $O = \emptyset$ then we return $true$ if any member of M' is present in σ and $false$ otherwise.

If neither base case holds, then, given that O is non-empty, we find a set, L which is the possible obligations which can be the last obligation to occur before t'. In order to do this, we find a set L_0 which is the set of all obligations with the latest start time. If there is a single unique obligation with the latest start time then L_0 will have a single member. If there is a tie, then L_0 will have multiple members. Either way, we will call the start time in question $L_0.t_e$. L is the set of all obligations which overlap $L_0.t_e$. Any obligation which does not overlap $L_0.t_e$ cannot be the last obligation to occur.

Next we define R to be the subset of L which only revokes permissions in M' and does not grant any. If R is empty, we check if any member of L_0 overlaps t'. If no member of L_0 overlaps t' then we return $true$ and terminate. If some member of L_0 overlaps t' then return $CheckDisjunction(O \backslash L_0, M', t', \sigma, t_0)$.

If R is not empty, then assume that all statements in R execute after all other statements in L. Because statements in R only revoke permissions and do not grant any

relevant permissions, we can order them arbitrarily. Let M_R be the set of permissions revoked by statements in R. All permissions in M_R can be assumed to be revoked when the obligations in R are executed. Hence, we know that a schedule which denies all of M' exists if we can order $O \backslash R$ such that all of $M' \backslash M_R$ is denied. As such, we return $CheckDisjunction(O \backslash R, M' \backslash M_R, t', \sigma, t_0)$.

Timing Analysis

The first step of the main algorithm will be carried out once for each obligation. Its substeps will be carried out once for each disjunction in the condition. Step 1(c)iv will invoke $CheckDisjunction$ up to twice for each pending obligation (if they all overlap b with both their start and end). As such, $CheckDisjunction$, is called $O(n^2m)$ times where n is the number of obligations and m is the size of the policy. When $CheckDisjunction$ runs, it scans through every obligation and every permission involved in each obligation, so its run time is $O(nm)$, not counting the recursion. In each recursive case, we remove at least one obligation before recursing. As such, the depth of recursion is bounded by the size of O. Since each call to $CheckDisjunction$ can result in at most one recursive call and the depth of recursion is bounded by n, the total running time for $CheckDisjunction$ is $O(n^2m)$. And hence to total running time for the algorithm is $O(n^4m^2)$.

Theorem 5 *Under the restrictions identified above, the problem of determining whether a concrete system is strongly accountable is in* $\boldsymbol{P}$.

4.2.2 Weak Accountability

Unfortunately, in the case of weak accountability, there is an additional complication. Our above algorithm, simplified to only check $t' = b.t_e$ would be sound and would identify many weakly accountable cases. However, it is not complete and would incorrectly identify certain accountable states as being unaccountable.

If a system is strongly accountable, and one obligation must wait to execute until another one has completed, then those two obligations must not overlap. In a weakly accountable system, such overlap is acceptable, so long as the end deadline for the dependent obligation falls after the deadline of the other obligation.

As such, our definition of weak accountability requires that we pay attention to such dependencies. If two obligations overlap, we cannot assume that they can happen in either order. The presence of dependencies may mean that only some schedules will be possible, and our algorithm above would mark some states as unaccountable despite the fact that the schedule which invalidates a given condition cannot actually occur.

It turns out, in fact, that in this case, unlike the situation with the three general assumptions, that it is more difficult to exactly determine accountability in the weak accountability case. Specifically, the problem of completely identifying weakly accountable states in this concrete model is Co-NP Hard. In the following subsection, we present a reduction of 3-SAT to weak unaccountability checking.

Reduction

Given a 3-SAT expression S with a set of variables $X = x_1, x_2, ..., x_n$, we construct the following system:

Let there be a set of permissions, $m_1, ..., m_n$ and also m^*. To simplify things, rather than write a policy and a list of pending obligations, we will assume that there is a single user carrying out all obligations and express the obligations in the form $(pre = condition, post = effect, [t_s, t_e])$.

The following pending obligations exist:

$o_{start} = (pre = true, post = Grant\ m^*, [1, 2])$

$o_{1,t} = (pre = true, post = Grant\ m_1, [1, 2])$
$o_{1,f} = (pre = true, post = Revoke\ m_1, [1, 2])$
...
$o_{n,t} = (pre = true, post = Grant\ m_1, [1, 2])$
$o_{n,f} = (pre = true, post = Revoke\ m_1, [1, 2])$

$o_r = (pre = true, post = Revoke\ m^*, [3, 9])$
$o_g = (pre = S', post = Grant\ m^*, [6, 12])$
$o_u = (pre = m*, post = nothing, [15, 20])$

Where S' is constructed as follows: Given a clause $(x_i \vee \neg x_j \vee x_k)$ from S, we replace it with $(m_i \vee \neg m_j \vee m_k \vee \neg m^*)$ in S'.

Then we ask whether or not this system is weakly accountable. If it is not, then there exists a satisfying assignment of variables for S.

The logic is as follows. Obviously, for $1 \leq i \leq n$, $o_{i,t}$ and $o_{i,f}$ will always be able to execute at their deadline. Likewise, o_{start} and o_r will. And, o_g will definitely be able to execute at its deadline, since the execution of o_r guarantees that S' will be true. However, whether or not o_u will be able to execute at its deadline depends on how o_r and o_g are scheduled.

If o_r happens after o_g, then m^* will not exist, and, as such, o_u will be unable to execute. As such, o_u will result in the state being unaccountable only when o_r can happen after o_g. Clearly the time limits do not prevent such a schedule. So the question is whether or not a dependency does. If there is no satisfying assignment for S, then S' cannot be satisfied until m^* is revoked, in which case, o_g depends on o_r and therefore cannot execute until after it. In that case, the system is accountable. If there is a satisfying assignment, then it is not guaranteed that o_g will occur after o_r and, as such, o_u is not guaranteed to be able to execute, and the system is unaccountable.

As such, the problem of checking accountability in a concrete system as described is Co-NP Hard. However, the model we have described is actually a superset of the original HRU access matrix model. The original model does not support either negative tests for permissions or the use of logical or to connect tests. We are still searching for an algorithm which would solve the weak accountability question in the more traditional version of the HRU access control matrix.

4.3 Accountability Checking with Uncontrollable Events

Until now, we have only discussed checking accountability in systems in which no uncontrollable events are present. This is because uncontrollable events pose additional challenges. The worst of these challenges is that, unless we know otherwise, we must assume that events can occur in any combination and order. The only assumption we make in the most abstract case is that no event can occur more than once in a given time click. As such, the number of possible states st'_i which follow from a state st_i after applying the events is

$|\mathcal{E}_U(st_i.t)|!$, that is to say, the factorial of size of the set of uncontrollable events which can occur at time $st_i.t$.

Even if we were to make an assumption that the order of the events is irrelevant, it would still leave us with $2^{|\mathcal{E}_U(st_i.t)|}$ possible successor states. Clearly, this is unmanageable. As such, there is no generalized solution for determining accountability in user obligation management systems with any substantial number of uncontrollable events. Instead, we propose several techniques which can be applied when we are able to make assumptions about the nature of the events.

4.3.1 Non-invalidation

Answering the general question of whether or not a system with uncontrollable obligations is accountable is quite difficult. However, it is substantially easier to describe a property which will ensure accountability.

First, let us assume that we have a user obligation management system which is accountable when the set of uncontrollable events is empty. In this system, only certain actions can possibly become obligations because there exists an obligation policy rule with a left hand side which is caused by a user action and a right hand side whose obligation function can create an obligation for that action. We call this subset of the actions $\mathcal{A}_B$. Now, let us consider a candidate set of uncontrollable events $\mathcal{E}_U$. If it is the case that this set of uncontrollable events can never change the state of the system in such a way that conditions of policy rules for these actions become unsatisfied, then the uncontrollable events can have no impact on the existing obligations. That is to say, the uncontrollable events cannot invalidate the preconditions for the obligations.

In order to have no impact on accountability, a second property would need to hold: if any uncontrollable event causes an obligation to occur then the preconditions of that obligation's action must be true for all reachable system states. Alternately put, the allowable actions in the system and the uncontrollable events cannot invalidate the preconditions of any new obligations caused by the uncontrollable events. As such, let us assume that there is also some candidate policy $\mathcal{Q}' \supseteq \mathcal{Q}$ which adds obligation policies for these new events. We can then define a set of actions which can become obligations as a result of uncontrollable events with that set of policies. We call this set of actions $\mathcal{A}_U$. If $\mathcal{A}_U$ cannot be invalidated by actions and events from $\mathcal{A}_B \bigcup \mathcal{E}_U \bigcup \mathcal{A}_U$ then we know that

obligations resulting from the events will always be able to happen.

We also need a third property, related to the first, which is that the obligations triggered by the events cannot invalidate the preconditions of other obligations. As such we replace our first property, which states that $\mathcal{A}_B$ cannot be invalidated by $\mathcal{E}_U$ with a more complete one which says that $\mathcal{A}_B$ cannot be invalidated by $\mathcal{E}_U \bigcup \mathcal{A}_U$

Determining whether or not non-invalidation holds between a system and a set of uncontrollable events cannot be done algorithmically in the abstract meta-model, but it should be able to be accomplished in particular concrete models.

For example, in the HRU-based model we outlined in section 4.2, we can perform this analysis by looking at the sets of permissions required by potentially obligated actions and comparing this against the effects of the uncontrolled events. Unfortunately, doing an exact analysis of whether or not the non-invalidation property holds would require determining the full set of reachable states of the system. This is impractical in most systems.

Instead, we can do an analysis which recognizes a sufficient condition for non-invalidation. Specifically, we examine preconditions for possibly obligated actions. For a set of preconditions, $Cond$, and a set of actions and events, E, if there exists no state, st, precondition, $cond \in Cond$, and action or event, $e \in E$, such that $cond(st) = true$ and $cond(e(st)) = false$ then we can know that E cannot invalidate $Cond$.

Thus, if we know that $\mathcal{E}_U \bigcup \mathcal{A}_U$ does not invalidate $\mathcal{A}_B$'s conditions we can know that the uncontrollable events will not invalidate the obligations. Checking this exactly would be difficult, but there is a sufficient condition which we can check easily.

Preconditions must be expressed in conjunctive normal form. As such, we can know that if there exists no pair of an action or event and a conjunct from a precondition such that one of the conjunct's terms can be invalidated by the action without validating another of its terms, then it cannot be possible for any action to invalidate any condition.

For example, if there is a precondition $(A \vee B \vee C) \wedge (B \vee \neg D)$, then an event which had the effect $revoke\ A$ would potentially invalidate this precondition. If the system could reach the state $A, \neg B, \neg C, \neg D$, at that state the precondition would be satisifed. But the effect $revoke\ A$ would invalidate the precondition. However, the effect $revoke\ A; grant\ B$ could never invalidate the precondition. Obviously unrelated effects, such as $grant\ E$ or $revoke\ D$ would also be unable to invalidate the precondition.

As such, we can check all pairs of disjunctions from preconditions with effects

from actions and uncontrollable events in order to know if we have achieved the sufficient condition. If we know that both of our non-invalidation properties hold, and that the original system was accountable, then it follows that the new one will be as well.

4.3.2 Predictable Events

Another possibility is that although uncontrollable, events may often be predictable. If an event is likely to happen once in a known time period, then we can create an obligation which represents that event. With that in place, analysis can continue as usual.

Our category of predictable events can actually be divided into two categories. The first are events for which the time period is predictable. For example, it might be that certain events are likely to happen during business hours. The second category are events which only occur once. If there is an event which only occurs once, then we can know that it is likely to occur once between now and infinity. As such, we can assign it as an obligation with a very large time period.

In either case, we can add the event to the set of obligations and run our accountability analysis. If we are not certain whether or not a given event will occur, we can handle this by running the analysis twice. However, if there are several events which may or may not occur, or which might occur multiple times in an interval, this process will quickly be overwhelmed, as it is exponential in the number of events involved. As such, it is definitely preferrable to be used only with events which we know will happen within some known time period.

In summary, there is analysis which can be applied in at least some cases, but there is no comprehensive strategy for dealing with uncontrollable events in user obligation management systems. It would be easiest to simply ignore uncontrollable events, but practical systems are likely to respond to outside events such as customer requests or problem reports by assigning obligations to users to handle them. As such, we decided that it is important to model them, even if we cannot offer comprehensive solutions for handling them.

4.4 Related Work

There have been other attempts to analyze systems with obligations to determine whether or not parties have sufficient rights to carry out their obligations. Firozabadi et al.

[14] describe a system for reasoning about obligations and policies in virtual organizations. However, their policies and obligations are both just static allotments of resources at specified time periods, making the comparison quite simple at the cost of being very specific to their model.

Kamoda et al. [26] attempt something a little more comprehensive in their model of policies in a web services setting. They model obligations and privileges which combine subjects, actions, and roles including a model of role hierarchies. However, their model is still limited because they model privileges and obligations as being triggered only by events which are independent of the actions in the system. As such they are unable to model any situation in which user actions can change the state of the system.

A large amount of work has been done on access control policies. A variety of policy languages and models have been proposed. Some of them are generic (for example, [54, 21, 22, 13, 43]) while others are for specific applications (for example, [10, 39, 48, 1]) or data models (for example, [2, 29, 3, 18]). A common problem in access control is compliance checking, that is, whether an access should be allowed according to an access control policy. Depending on access control models, compliance checking may be very simple (such as checking an access control list), or quite complex (as it is in distributed trust management [8]). The problem of determining whether a state is accountable is analogous to compliance checking in access control, in that it must be performed to determine whether to allow a requested operation. However, since it needs to consider the fulfillment of obligations in future states, the determining accountability is inherently more complicated.

Another important class of problems in access control is static policy analysis, for example, safety analysis [20, 44] and availability analysis [30]. It is interesting to investigate what types of policy analysis can be performed on obligation policies, but existing work does not address obligations. The analysis presented in this paper is dynamic, but accountability can be used in static analysis of systems.

Chapter 5

Blame Assignment

In this chapter, we describe the functioning of the blame assignment module. We make the assumption that the system is attempting to maintain an operational policy which guarantees at least accountability. The system may have a stronger operational policy. For instance, it might ensure that certain important obligations will still be carried out even in the event of some failures. However, we assume that at very least, the operational policy attempts to provide accountability. It should be noted that because the success of the operational policy depends on the voluntary actions of users, it cannot be guaranteed.

5.1 Blame Assignment

An aspect of obligation systems which we wish to automate is the assignment of blame. When an obligation failure occurs, we wish to know who is to blame for that failure. The first question to examine is whether or not the user to whom the obligation was assigned possessed the ability to carry out the obligation. If he did, then it is clearly his fault that he failed to carry out his obligation. If, however, he did not, it is likely not his fault.

One possibility is that the fault lies with the system. If an obligation is assigned which a user is unable to fulfill, this may be because the system failed to adequately ensure that needed permissions would be available to the user. However, if the system achieved an accountable state at some point between the assignment of the obligation and the obligation failure, then we know that this cannot be the case. If the system did achieve accountability

then the fault lies with some other users for failing to fulfill their own obligations.

Failing to fulfill an obligation can potentially result in other users lacking needed privileges. As such, a single failure can sometimes result in a cascade of obligation failures and some of the failures can have quite serious consequences. However, modeling and understanding the dependencies between different obligations turns out to be quite difficult.

In traditional (that is, non-automated) obligation systems, the assignment of blame for failures is often carried out by examining the events surrounding the failures and doing a postmortem analysis. Such an analysis often factors in a variety of facts concerning responsibility such as who was assigned a task, who was able to do it, whose job it was to do it, who was expected to do it, and what communication there was concerning the task. Ideally we would do a similar postmortem analysis in our automated system. But simple information about which obligations were assigned to which users is not going to be sufficient. Given a series of failed obligations, an automated tool may be able to determine which obligations, had they been fulfilled, would have satisfied the preconditions of other failed obligations. However, this information alone is not adequate to determine blame.

Example 1 *Let us consider, for instance, a situation in which there are three users, Alice, Bob, and Carol such that each of these users has an obligation, and Carol cannot fulfill her obligation unless at least one of Alice or Bob fulfills theirs. If neither Alice nor Bob fulfill their obligations and, as a result, Carol fails hers, are Alice and Bob both equally at fault? The answer depends on the circumstances. If Alice's job was specifically to make sure that Carol has what she needs, but Bob's task only enabled Carol as an incidental side-effect, then the responsibility would fall more on Alice. If, instead, Alice was known to be very busy and Bob was given the task in order to ensure that it got done even if Alice could not do it, then the responsibility would fall more on Bob. Further, if Bob had told Alice that he would do it and that she did not have to worry about Carol failing if Alice failed her obligation, then clearly the blame for Carol's failure would be Bob's. As such, it is clear that issues of responsibility and blame are more complex than simply determining if an action will cause a precondition to be satisfied.*

Any postmortem analysis of the responsibility for failures would need to include information about the policies, intents of the policies, communication between parties, and other factors. Such analysis is certainly possible, but it would be very difficult, if not

impossible to automate.

As such, we instead propose solving the basic problem of determining responsibility by tracking responsibility in an active, on-line fashion rather than attempting to determine it after the fact. Because responsibility is tied into the intent behind the assignment of obligations, we propose a policy-driven system for responsibility tracking. Instead of attempting to discover, after-the-fact, who could have prevented the situation, we wish to keep track ahead of time of what the consequences of an obligation failure are.

What we propose is a module in an obligation system which keeps track of which obligations bear responsibility for which other obligations. Essentially, the module will keep a directed graph of responsibility, indicating which obligations are responsible for which other obligations. As obligations are fulfilled and discharged, and as other circumstances change, this module should update the graph to reflect changes in responsibility. The responsibility information in the module will serve both as a means of determining blame after the fact and before the fact. Users can consult the module to better understand the implications of their actions in case there are circumstances which may lead them to have to do things such as deciding which of two obligations they have time to fulfill.

Because, as we demonstrated above, there are a variety of different possible intents behind the assignment of obligations, we wish to allow for a policy which describes what the assignment of responsibility should be. In practice, we would like to have policies which consist of one general rule and some special exceptions. That way instead of having to imagine every possible situation and write a policy for it, the administrator could choose a sufficient default policy and then only worry about specifying more specific policy rules for exceptional cases.

5.1.1 Desirable Properties

Although we have argued that blame assignment is not simply about knowing which obligations enable which other obligations, the dependencies between obligations are important to determine blame. Namely, if two obligations are not at all dependent on each other, then the failure of one should not be blamed on the failure of the other in any circumstance. Similarly, if one is completely, directly, and solely dependent on another obligation, then clearly the failure of the later one should be blamed on the failure of the earlier one.

These two properties effectively form an upper and lower bound for relationships between pairs of obligations. We do not expect that the majority of pairs of obligations will have one of these two properties. Rather we expect that most will live in the great grey area in the middle, where responsibility is unclear. This is why we later describe the specifics of a policy-driven blame assignment engine. But for pairs of obligations which do have one of these two properties, obviously our system should properly assign responsibility or the lack of it. We say a responsibility assignment policy is *valid* if the above two properties are preserved.

As such, we wish to formalize the two properties so that we can prove that a given policy is at least valid. For this purpose, we introduce the following notations to facilitate our discussion.

Let us assume that we have some set of obligations B, a start time, t_0, and a set of possible future times, T, such that for all $t \in T, t > t_0$. For convenience below we augment each obligation $b = obl(s, a, O, t_s, t_e) \in B$ with a version of the function $\mathsf{permitted}$, denoted $b.\mathsf{permitted}$, that is specialized to b. In this analysis, we assume that $E_U =$. That is, we assume that there are no uncontrollable events. As such, the type of this function is $b.\mathsf{permitted} : \mathcal{FP}(\mathcal{S} \times \mathcal{A} \times \mathcal{O}^*) \times \mathcal{ST} \rightarrow \{true, false\}$. Given a set of actions AP to be executed in state st such that $(b.s, b.a, b.O) \in AP$, $b.\mathsf{permitted}(AP, st) = \mathsf{permitted}((b.s, b.a, b.O), AP, st)$.

Definition 9 *A* schedule *of obligations in B is a function $H : B \rightarrow T \bigcup \{null\}$ in which $H(b) = t$ means that b is performed at time t. If $H(b) = null$ for some obligation $b \in B$, this means that in schedule H, o is not fulfilled.*

We also define a helper function $AP(H, t) = \{b | H(b) = t\}$. In our obligation system, all transitions are deterministic. As such, given a start state, st_0 at some time t_0, any schedule H over B uniquely defines a sequence of states $st_H(t)$, defined by letting the action plan at any time t be $AP(H, t)$. Note that if an action plan includes obligations whose conditions are not met, the system will reject them, and as such, they will not affect future system states.

A schedule is considered to be *valid* for B, st_0, and t_0 if $\forall b \in B$, $b.\mathsf{permitted}(AP(H, b), st_H(H(b))) = true$ and $b.t_s \leq H(b) \leq b.t_e)$ or $H(b) = null$. Intuitively, in a valid schedule, the action of each obligation is authorized to be performed at the

scheduled time.

Given two schedules, H_1 and H_2, where H_1 is defined over some set of obligations B_1 and H_2 is defined over some $B_2 \subseteq B_1$, we define $H_1 \sim H_2$ to be the schedule H' such that $H'(b) = H_2(b)$ when $o \in B_2$ and $H'(b) = H_1(b)$ otherwise. For example, suppose we have two schedules $H_1 = ((b_1, 10), (b_2, 20), (b_3, 30), (b_4, 40))$ and $H_2 = ((b_2, 35), (b_3, 25))$. Then $H_1 \sim H_2 = (((b_1, 10), (b_3, 25), (b_2, 35), (b_4, 40))$.

For convenience, we also denote $H^{B,t}$ to be the schedule defined over B such that $H^{B,t}(b) = t \forall b \in B$. That is, given a set of obligations B, $H^{B,t}$ is the schedule which says that all the obligations in B happens at time t.

Not Responsible For

The first property we call *not responsible for*, which captures the concept that obligations are completely unrelated to each other.

Formally, we say an obligation b_1 is *not responsible for* another obligation b_2 at some time t_0, if given any valid schedule H over B such that

1. (1) $H(b_1) \neq null$; and
2. (2) for $H' = H \sim H^{\{b_1\},null}$, it is the case that $\forall t', b_2.t_s \leq t' \leq b_2.t_e$, $\neg b_2.\mathsf{permitted}(st_H(t'), AP(H, t')) \vee\ b_2.\mathsf{permitted}(st_{H'}(t'), AP(H, t'))$.

To summarize in a plainer language, b_1 is not responsible for b_2 if there is no valid schedule H, in which b_1 happens, such that b_2 can happen in H, but such that it cannot happen in an alternate schedule H' in which b_1 does not occur. Specifically, we compare H to a schedule just like H except that we remove b_1 from it, which means that b_1 does not occur and neither does any obligation whose condition is invalidated by the removal of b_1 from the schedule. So, if there exists any schedule for which whether or not b_1 happens has a negative outcome on whether or not b_2 can happen, then we do not say anything about b_1's responsibility for b_2. But if no such schedule exists, then we say that b_1 is not responsible for b_2.

Definitely Responsible For

There also exists the converse idea. If an obligation is always fundamental to another obligation, then clearly it should be responsible for that obligation.

Formally, we say that an obligation b_1 is *definitely responsible for* another obligation b_2, if both of the following hold:

1. For all valid schedule H defined over B such that $H(b_1) \neq null$, let $H' = H \sim H^{\{b_1\},null}$. Then $\forall t', b_2.t_s \leq t' \leq b_2.t_e$, we have $b_2.\mathsf{permitted}(st_{H'}(t'), AP(H', t'))$ is false.

2. There exists some valid schedule H defined over B where $H(b_1) \neq null$, such that $\forall t', b_2.t_s \leq t' \leq b_2.t_e$, we have $b_2.\mathsf{permitted}(st_H(t'), AP(H, t'))$ is true .

To summarize in a plainer language, an obligation b_1 is definitely responsible for another obligation b_2 if there is no way that b_2 can happen when b_1 does not happen, and there is at least some case in which b_2 can happen when b_1 does. The second condition is needed because we do not want to blame b_1 for b_2 in circumstances in which there is no way that b_2 can occur.

5.1.2 Blame Assignment

Because we wish to discover who was at fault when an obligation failure occurs, we need to keep track of which obligations are responsible for enabling which other obligations. In order to do so, we represent responsibility using a directed graph. Each node in the graph corresponds to an obligation. There is an edge in the graph from obligation b_1 to obligation b_2 if and only if b_1 is considered to be responsible for b_2.

If our assignments of responsibility are reasonable, then it should be the case that any obligation depends only on obligations which are before it. In this case what we mean by one obligation being after another depends on the type of accountability that our system is attempting. If the goal is only weak accountability, then b_1 is considered to be before b_2 if $b_1.t_e < b_2.t_e$. If the goal is strong accountability, then b_1 is considered to be before b_2 if $b_1.t_e < b_2.t_s$. It is worthwhile to note that if this property holds, then it will be the case that our graph is acyclic. This is a desirable property since if our graph contains cycles, we could wind up blaming an obligation's failure for causing itself to fail, which is not quite sensible.

When there is an obligation failure, we can then use this graph to determine who was responsible for the failure. In order to do so, we use an algorithm which traces

backwards following the links of responsibility in reverse, in order to figure out which failed obligations are to blame for this failure.

The algorithm uses a working set, K, which contains obligations which are potentially to blame and produces an output set, L, of obligations which are to blame. The initial failed obligation is designated as b. The algorithm assumes the existence of sufficient logs to check things such as what the system state, st, was at different times, which we will designate as $st(t)$, and whether or not particular obligations were carried out. The action plan which was executed at any particular time t is represented as $AP(t)$. And it should be noted that the algorithm operates slightly differently depending on whether or not the system is attempting to maintain strong accountability or weak accountability. This difference is important because it changes expectations about how a user is expected to behave if an attempt to take an action is denied.

1. $K := \{b\}$
2. $L := \emptyset$
3. If K is empty, terminate.
4. Select an obligation b' from K.
5. $K := K - \{b'\}$
6. Using the system logs, check when $b'.\mathsf{permitted}$ was true.
   (a) If in a weak accountability system, if
   $b'.\mathsf{permitted}(st_{b'.t_e}, AP(b'.t_e) \bigcup \{b'\})$
   is true, then $L := L \bigcup \{b'\}$ and go to step 3.
   (b) If in a strong accountability system, if
   $b'.\mathsf{permitted}(st_t, AP(t) \bigcup \{b'\})$
   is true for all t such that $b'.t_s \leq t \leq b'.t_e$, then
   $L := L \bigcup \{b'\}$ and go to step 3.
7. Let F be the set of all obligations which have edges which point to b' and which failed.
8. $K := K \bigcup F$

9. Goto step 3.

In essence, we simply move backwards through the graph, looking at each obligation which could be responsible. For each obligation, if its condition was true, then it must be the case that it was simply not done by the user to which it was assigned. As such, we need not seek anyone further to blame for it. However, if its condition was false, then the access control policy would prevent it from happening, thus it is not the fault of the user to which the obligation was assigned. Instead, we must look to the obligations which are responsible for that obligation and see which of those failed, and, in turn, determine whose fault that was.

It should be noted that it is possible that in some cases the algorithm above could return an empty L set, indicating that no one is to blame. For instance, if an obligation did not have the permissions it needed to happen, but all of the other obligations responsible for it occurred. At first, this could be seen as a failing of the analysis, but instead it is an indicator which tells us that the system made a mistake. Most likely this will occur when there is an obligation in the system which is simply not able to be completed.

Even if a system strives to maintain an accountable state, it may be the case that obligation failures push it out of that state. And then difficult decisions may need to be made weighing the goal of returning to accountability against the overall goals of the obligation policies. This may potentially result in the assignment of obligations which cannot be fulfilled or obligations which interfere with the fulfillment of other obligations.

An empty L set could also indicate that the responsibility analysis has not identified all obligations which should have been considered responsible. As we will outline in the next section, the assignment of responsibility is a task which depends on the specifics of a system, and as such cannot be described only in terms of the meta-model. Generally speaking, we expect that responsibility assignment will not be algorithmically difficult, but there are theoretical systems for which it could be quite difficult. As such, there may be some situations where responsibility assignment would be approximated, resulting in occasional mistakes in exchange for greater efficiency.

5.1.3 Responsibility Assignment

Given a set of obligations, we wish to analyze them and form some assignment of responsibility. There is a great deal of flexibility in assigning the responsibility, but there are several properties which the assignment should meet. The first two are based on our properties above. If, at the time that we are assigning responsibility, b_1 is not responsible for b_2 according to the definition presented in the previous section, then b_1 should not be assigned responsibility for b_2. If, at the time that we are assigning responsibility, b_1 is definitely responsible for b_2 according to the definition presented in the previous section, then b_1 should be assigned responsibility for b_2.

However, the reverse may not be true as it is possible that for one obligation there is no obligation definitely responsible for it. For instance, in example 1, since both Alice's and Bob's obligations can enable Carol's, according to our definition, it is not hard to see that neither of them are definitely responsible for Carol's obligation. In this situation, depending on the policy, the system may choose one of Alice and Bob's obligations to be responsible for Carol's.

As a result, there are likely many different ways to assign responsibility which meet all three of the properties outlined above. In this paper we are going to present one algorithm which can be adjusted to form a variety of different basic policies. Unfortunately, the first step of this algorithm is not something which can be generally applied to any system which fits the meta-model. Instead, it must be done in a system-specific way and there do exist specific systems for which the step cannot be done. However, it is our belief that the step can be accomplished in many practical systems in reasonable time. To back-up our argument, we describe how to perform this step in a concrete example system in the next section.

This first step is as follows: given a set of outstanding obligations B and a particular obligation b, find a set of subsets of B, which we will call $N(b)$, that has the following properties:

1. In any schedule for which at least one member of each set occurs, then b will be permitted.

2. There exists a schedule in which all the obligations from any one set do not occur, but all other obligations occur, in which b is not permitted.

Once we have a set of subsets of B which has such a property, then we are going to chose precisely one member of each subset to be considered responsible. Because there may be some subsets which overlap, it should be noted that the number of subsets only forms an upper bound for the number of responsible obligations. There are a number of different ways in which the particular member can be selected, and these reflect different policies. We will detail the specifics of these later, but first we will prove that as long as one member from each of these sets is chosen then our properties will be met.

The first property is that any obligation, b_1, chosen as being responsible for another obligation, b_2, must not have a relationship of "not responsible for" with b_2. Let us assume that some member, b_1, of one of the subsets, x_1, has such a relationship, then for all schedules, whether or not b_1 occurs, the permission status for b_2 is not positively affected. By the second property of $N(b_2)$ there exists a schedule, H, in which all obligations other than those in the set x_1 occur and b_2's can occur. If we define $H' = H \sim H^{\{b_2\},t}$ for some $t|b_2.t_s \leq t \leq b_2.t_e$, we can see that by the fact that b_1 is not responsible for b_2, since b_2 could not happen in H, it also cannot happen in H' for any choice of t. However, this violates the first property of $N(b_2)$, which says that any schedule in which at least one obligation in each set from $N(b_2)$ occurs results in b_2 being able to occur. Therefore we have a contradiction, so our one assumption must be false. Thus, we know that if we have minimal subsets, then all members of the subsets must contain only obligations which are not "not responsible for" b_2.

The second property is that any obligation, b_1, which has a relationship of "definitely responsible for" b_2 must be chosen as being responsible. However, it does not need to be chosen directly as being responsible for b_2, since it could be responsible for another obligation which is responsible for b_2 or so forth. Rather, it needs to be the case that there exists a path from b_1 to b_2 in the graph.

At this point, we note that there is not a single unique set of minimal subsets which has the listed properties. If, for example, there are three obligations, b_1,b_2, and b_3, and b_3 depends on b_2 which, in turn, depends on b_1. The set $\{b_2\}$ should definitely in $N(b_3)$, but $\{b_1\}$ is optional.

However, it should be the case that only subsets which have some form of dependence relationship are optional. To formalize this idea, we're going to define a new relationship *needs*. We say that an obligation b_1 needs a set of obligations x_2 if $x_2 \in N(b_1)$.

Further, we say that a set of obligations x_1 needs another set of obligations x_2 if for every $b_1 \in x_1$ either $b_1 \in x_2$ or there exists a subset x_2' of x_2 such that b_1 needs x_2' and for all $b_2 \in x_2$ either $b_2 \in x_1$ or there exists a subset x_2' of x_2 such that $b_2 \in x_2'$ and there exists $b_1 \in x_1$ such that b_1 needs x_2'. We then define N^* to be the transitive closure of the needs relation over the power set of B.

Then, the desired property is that if b_1 is definitely responsible for b_2 then $\{b_1\}$ is in $N^*(b_2)$. So long as this is the case, then it follows that their must be a path of responsibility chosen between b_1 and b_2. This we prove by induction.

Our inductive hypothesis is that if a set of obligations x is n steps away from b_2 using the need relationship, then for any set x' such that x needs x', then there is a path from at least one of the member obligations of x' to b_2 in the responsibility graph.

Our base case is the situation where b_2 needs x', that is, when they are one step apart. If b_2 needs x' then $x' \in N(b_2)$ and our algorithm dictates that we chose at least one node from each set in $N(b_2)$, so the base case is established.

In our inductive case, we assume that there exists some x which is n steps from b_2 using the need relationship and that there exists a path from at least one node in x to b_2 in the responsibility graph. Let b_1 be the node in x which has a path to b_2. Let us be given some x' such that x needs x'. Let us examine how b_1 may be related to x'. If b_1 is in x' then we have proven our inductive hypothesis and are done. If b_1 is not in x' then it follows that b_1 needs some subset of x', which we will call x_1' Since this is the case, it means that $x_1' \in N(b_1)$. This, in turn means, that by our algorithm, we must select some member of x_1' to be responsible for b_1. As all member of x_1' are also members of x' and there is now a path from this member to b_1 and from b_1 to b_2, we have established the inductive hypothesis, and our proof is complete.

Now we need to prove that if b_1 is directly responsible for b_2 then it is actually the case that $\{b_1\}$ is in $N^*(b_2)$. If we assume that $\{b_1\}$ is not in $N^*(b_2)$, then it follows that no set in $N^*(b_2)$ needs $\{b_1\}$. If no set in $N^*(b_2)$ needs $\{b_1\}$, then it follows that there must exist at least one member of each set in $N^*(b_2)$ which can happen even when b_1 does not happen. Therefore, if we consider the schedule in which all of those members happen and b_1 does not happen, by the definition of N, it follows that b_2 must be able to happen. Hence, there exists such a schedules, and b_1 cannot be definitely responsible for b_2. This is a contradiction, so our only assumption, that $\{b_1\}$ is not in $N^*(b_2)$, must be incorrect.

Hence we have proven what we set out to.

Therefore, we know that if we apply our algorithm, our responsibility assignment satisfies our properties. In section 5.2, we demonstrate a simple system in which it is possible to follow our algorithm because the first step can be accomplished efficiently.

Choosing Obligations to be Responsible

As we have shown above, in our algorithm there is a lot of room for flexibility concerning how we select the specific obligation which is considered to be responsible. So long as at least one obligation from every set in $N(b)$ is selected, then the system is guaranteed to satisfy the principles outlined.

As such, we would like to now discuss several different standard alternatives which could be used to form a basic policy. It should be noted that the sets in $N(b)$ are not guaranteed to be disjoint. Generally, speaking, it seems wise to avoid giving responsibility to any two obligations which are in the same set. As such, one possible policy is to chose a minimum set of obligations such that at least one obligation from each set is selected. It should be noted that in the worst case, minimum set cover is an NP-Complete problem, and so in difficult cases, this could simply be approximated instead. Alternately, if responsibility should be widely distributed, a maximum set could be chosen.

Further, additional simple policies involve using the fields of the obligation to decide on a responsible party. There maybe circumstances where you would want to make the responsibility fall on the obligation in a given set with the earliest end time or ones where it should fall on one with the latest end time. Alternately, other information concerning characteristics of the subject, objects, or actions could come into play. Giving responsibility to the most senior user from each group or the highest ranking, might often be a desirable policy. Other options include choosing the most important action, given some ranking of actions or the highest priority object. Any such policies are acceptable and available to use as default policies on which a more complex policy may be written.

5.2 Concrete Example

Let us now consider how blame assignment would work in a more concrete system. Specifically, we will use the same Access Matrix Model-based system which we introduced

in chapter 4.

5.2.1 Responsibility Assignment

In order to run the responsibility assignment algorithm for an obligation b, we need to be able to compute $N(b)$. Here we present the algorithm for doing so. Our condition for a given obligation is in conjunctive normal form. Each conjunct (that is, each disjunction) logically corresponds to a set in $N(b)$, and that is, in fact, how we build our sets.

For each conjunct, which we consider to be a set of permissions, we first check to see if the conjunct is guaranteed to be true under any schedule. If it is, then it does not need a set. The first step is to see if any permission and its opposite are both tested for in our conjunct. If they are, then we are done, since $m \vee \neg m$ is a tautology. Assuming that this isn't the case, then we have some distinct set of permission tests.

In what follows, we treat each permission test as being a positive test. We do this without loss of generality since we can convert any negative test to a positive test by reversing its presence in the current state (that is, adding it if it is not there and removing it if it is) and changing all grants into revokes and vice versa. So, for simplicity, we treat all tests in a given conjunct as positive, and hence represent the tests as a set of permissions, $M = \{m_1, ..., m_n\}$, but it is not the case that we are actually assuming them to be positive.

Since each test is distinct from the others, our conjunct is only guaranteed to be true if at least one of the needed permissions already exists in the system, and no existing obligation can remove it without adding another permission we need. So, in order to find this, we simply check our needed permissions against existing permissions in the system. If none of our needed permissions are already present, then we know that the truth of our conjunct cannot be guaranteed, so we move to the next step which is described in the next paragraph. If we find some of our needed permissions to be present, then we check for obligations which would revoke them which come before b or overlap b. If we do not find any such obligations, then we know that our conjunct is true, and hence, does not need a set in $H(b)$, and we move on to the next conjunct. If we do find any such obligations which do this then we have to examine whether or not they grant another permission which we need. If they do not, then we know that our conjunct is not guaranteed to be true, and we go to the next step. If they do, then we have to repeat these checks for the new permission, considering potential revoking obligations which overlap with our granting obligations or

which fall between them and b. If we find any sequence of obligations which can result in needed permissions being revoked without corresponding ones being granted, then we are not guaranteed, and we continue to the next step, constructing our set for $N(b)$.

Given our set of obligations B and our set of needed permissions, we first exclude any obligations which overlap with or come after b. We then define an output set, which we'll call N' and a candidate set which we'll call C. Both sets are initially empty. Then we examine each obligation which grants a needed permission. If no obligation which revokes that same right overlaps it or comes after it, then we add the obligation to N'. If not, then we examine the obligations that might revoke the right it needs. If all such obligations also grant a right needed, then we add our obligation to C, and note which obligation or obligations interfere with it (that is, overlap it or come afterwards and revoke the granted permission).

Once we have completed this process for all permissions, we revisit our candidate set. If there are any obligations in C which are being interfered with by obligations which are not in $C \bigcup N'$, then remove we them from C. We repeat this process until all obligations which remain in C are interfered with only by other obligations in $C \bigcup N'$. Then our final set is $C \bigcup N'$.

We know that any set created using this process has the two needed properties. Firstly, we know that if any one obligation in our set occurs, then our condition will be satisfied. Secondly, we know that if all the other obligations happen and all of the ones in the set fail to occur, then there is a schedule for which the condition for b will not be satisfied. This schedule is the one in which any grants in obligations which we did not select happen prior to the revokes. We know that this is a valid schedule because every grant which came after the last revoke is in our set and because when a system is strongly accountable, any two overlapping obligations must be able to happen in either order.

5.3 Related Work

There seem to be few papers directly related to the problem of blame in user obligation systems or other similar systems, however, questions of blame have been considered in other cases. For instance, blame assignment is related to auditing systems, which collect audit data and analyze them to discover security violations [42]. A majority of works in

auditing target intrusion detection, which is distinct from the problem addressed in this paper.

Obligations and contracts are central concepts in collaborative multi-agent systems. Deontic logic [34] is commonly used to express the agreed obligations among agents. Works in this area typically do not concern the interaction between obligations and the policies which produce them. Instead, they focus on expressing the dependencies between obligations, assuming such dependencies have already been identified [11, 19]. In this paper, we formalize the key properties for correct identification of dependencies between obligations.

Chapter 6

Failure Feedback

When we are examining a system in which both security policies and obligation policies are being enforced, this introduces new difficulties and subtleties. In particular, maintaining accountability often requires making decisions not merely based on the current access control state of a system, but based on possible future access control states. As a result, decisions about which actions should or should not be allowed become more complex. And, as they become more complex, they become more difficult for users to understand.

Although research on access control policies rarely talks explicitly about the needs of users to understand the policy, there is an implicit assumption that users have at least a basic understanding of the boundaries of the access control policies of the system which they are using. Sufficiently complex policies break this assumption. As such, we need a system to aid the user in being able to achieve his objectives. In this chapter, we describe such a system.

6.1 Problem

Computer systems exist to serve the needs of their users. In a multiple user system, one aspect of this is to ensure that users cannot interfere with each other or otherwise abuse the system. As such, systems have security policies. However, the underlying aim of the computer system is still to enable users to achieve certain goals. If a system is to remain usable it is necessary for users to be able to understand how to achieve these goals.

Sometimes a security policy will disallow an action which a user wishes to take.

This does not necessarily mean that taking the action is impossible, it may simply mean that it cannot be taken under the current circumstances. For example, there may be restrictions on what actions can be taken at what times or how many users can make use of a resource simultaneously or sometimes preapproval for certain actions may be required. In most usable systems, it is implicitly assumed that users can at least basically understand the circumstances under which they can take actions. Even if they do not have a complete knowledge of the security policy, they can find means by which to do the actions which they desire when such actions are possible.

Unfortunately, complex operational policies, such as accountability undermine that assumption. The more complicated the interaction is between the system state and the policies of the system, the more difficult it will be for users to know the circumstances under which their actions will be allowed. If we wish to use obligation policies, we need to find ways to ensure that the system is usable.

One possible means of doing this would be to attempt to explain to the user as fully as possible the specifics of why their action attempt was denied. This, for example, is the approach taken by Kapadia and Sampemane in [27] for explaining rejections in a system with policies which vary depending on time and location. In their model, they have policy rules which directly govern the use of resources. When a user's action is rejected, their system examines policies and circumstances to determine what part of the policy should be shown to a given user and examines meta-policies to determine what part of the policy it is allowed to show the user. Together it uses this to explain to the user, as best it can, the reason for the rejection.

However, in an obligation system, this sort of approach is likely to be very difficult, if not impossible. First, the inherent complication of the situation alone will cause difficulty. An action could be denied because it changes the effect of some other later obligation which in turn changes the effect of another obligation which causes a third obligation to no longer be guaranteed to occur. Explaining this to the user in a meaningful manner is going to be quite difficult. A second issue is that the explanation is going to involve not only revealing the policy to the user, but also revealing specific information about the state of the system and the obligations of other users. The user may not be entitled to know this information due to security or privacy concerns.

Even if we can explain to the user precisely why his action was disallowed, he will

not necessarily be able to figure out what he should do differently in order to successfully take the action he attempted. However, we would like to suggest that the purpose of explaining to a user the reason why their action was denied is so that they can formulate an alternate plan to achieve their goal (if possible). Because of the inherent difficulties in making such explanations in a system which is maintaining accountability, we propose that instead of explaining the reason for the rejection, we skip over this step and instead present to the user possible plans of action which will achieve the user's goal.

In this case, we do not have a precise idea of what the user's actual goal is, so in order to approximate it, what we attempt to do is to find a set of circumstances under which the user could carry out the same actions. This should be likely to achieve a similar result to the user's original actions.

6.1.1 Model

In order to make a attempt at tackling this problem, we are going to first make a small number of simplifying assumptions. In future research, we hope to relax these assumptions, but even a partially simplified version of this problem has not previously been examined in the literature.

Specifically, we are going to model a simplified system in which user actions do not trigger obligations as side effects. Instead, we are going to assume that obligations are going to arise in two manners. The first is that obligations may be the result of outside events. The second is that users who wish to do so may request that the system assign them obligations to do certain actions. That is, users can put in a request to do some set of actions in the future. If the system accepts this request then those actions become obligations. If the system denies the request, then, of course, they do not.

For the sake of simplicity, we also assume that the policies are designed in such a way that if the system is accountable, no event will cause obligations which make the system unaccountable. Actions requested by users could still make the system unaccountable if allowed, but we have previously shown how to recognize unaccountable states, so this can be avoided by denyng the action request.

6.1.2 User Feedback

A user may submit a schedule of desired actions which they intend to carry out and time periods for those actions. If the system accepts this schedule of actions, then these actions and their associated time periods become obligations for that user. If the system rejects these actions then it will attempt to formulate an alternate plan.

The situation in which a user attempts an immediate action is just a special form of this same process where a user attempts to schedule an action for the current time and then also carry it out. As such, we do not treat this any differently than an attempt to schedule further in the future.

The plans we generate must have several properties:

Property 1 *They must include the same set of actions which the user had in his schedule.* Although this will not guarantee that a plan will meet the user's goals, we feel that it will make it likely to.

Property 2 *The plans must, if accepted, leave the system in an accountable state.* Obviously, plans which do not result in accountability should not be generated by a system which is trying to preserve accountability.

Property 3 *Plans should not modify existing obligations.* Modifying obligations is a special task which should only be available to administrators and not ordinary users. As such, there is no point in presenting plans which modify obligations to ordinary users for approval.

On possibility is that, a plan might only assign actions to the user who attempted to schedule the initial actions. For example, the plan might involve the user taking actions which give him additional privileges or adding obligations which mitigate the effect of his actions. In this case, once the system has generated an alternative plan, the user has the option of accepting that plan. If accepted, the actions in the plan become obligations with the given time periods.

However, plans may also involve actions for other parties as well. For example, if the user needs his manager's authorization to take an action, the system should return a plan which includes his manager giving her authorization. In the event that other users are given actions to do in the plan, then those users must also agree to the plan before the

system will convert the plan into user obligations. We do not cover in this thesis the precise mechanism for how such an agreement would be reached and validated by the system, but simply assume that such a mechanism exists.

In summary, we wish to create a failure feedback module such that when a user submits a set of actions and time frames for those actions the module will return one or more plans which contain both the user's desired actions and the existing obligations and if accepted will result in the system being accountable. Our new plan can have additional actions and the users actions in it may be scheduled to happen at different times or in a different order.

We allow for actions to happen in a different order because sometimes the order of the actions requested may be the reason that the user's request was denied. Our expectation is that if the order of the user's actions is important then this will often be reflected by the access control policy. There may, however, be some systems where the effect of actions changes substantially depending on the order in which they are carried out. In such a system, it would be preferable to maintain the order of the users actions. Although we have not implemented this in our testing, the changes needed to maintain the order are described in section 6.2.2

6.2 Approach

In order to create such plans, we propose applying techniques from artificial intelligence. Specifically, we propose using a modified partial-order planner in order to generate plans. Partial-order planners are artificial intelligence planners which specifically attempt to generate plans where the actions in the plan are ordered according to a partial order. This is desirable because, although their model of time is not identical to what is used in our obligation system, it is possible to convert between the two models. We will describe the specifics of this conversion later in this section. Also, due to how the partial-order planning algorithm works, we can modify the planner such that it will respect our obligations.

Other than partial-order planners, several other types of artificial intelligence planners also exist. So, an obvious question is why use a partial-order planner in particular? The use of a partial-order planner is somewhat unusual since partial-order planners are no longer considered the state-of-the-art in the artificial intelligence community. Most planning

research currently focusses on GraphPlan[9]-based planners and SatPlan[28]-based planners which are much faster than partial-order planners.

GraphPlan-based planners generate a specialized graph which represents the sequences of actions available and annotates the graph with information about conflicts between actions. SatPlan-based planners transform planning problems into satisfiability problems and then use heuristic-based satisfiability solvers to solve the transformed problems. Both of these approaches are significantly faster than traditional partial-order planners for most problems. These two approaches may also be combined to form a hybrid approach.

As such, they are generally preferred in the planning community. However, there are also some downsides to these approaches, as described by Nguyen and Kambhampati in [36]. The first is that they tend to produce plans which are less efficient. That is, the plan produced by GraphPlan or SatPlan-based planners generally have more actions in them than the ones produced by partial-order planners. Since actions in our system will have to be carried out by users, it is certainly desirable that the plan be shorter than longer.

The second advantage of a partial-order planner is that the plan it produces is a partial order of actions. Specifically, they aim to find the least amount of ordering constraints necessary. Satisfiability-based and GraphPlan-based planners instead create a plan with a list of steps in which each step contains a set of actions which can be carried out in any order. This is not a total order, but it is also not an arbitrary partial order. As such, they do not map as well to obligation systems in which obligations are not likely to be arranged into steps and can, instead, overlap arbitrarily. Therefore it is difficult to translate existing obligations into the time representations used in GraphPlan-based and Satisfiability-based planners.

Partial-order planners also support the creation of custom heuristics. Currently, the planner attempts to find the shortest plan, but the heuristics could encourage it to first examine plans which are minimized according to some other heuristic, such as least cost of actions or fewest users involved. We do not explore the creation of or gauge the effectiveness of such heuristics in this thesis, but the possibility it there.

The last important difference is that it is unclear how to modify the basic algorithms of GraphPlan or SatPlan to guarantee that certain actions will be included. If we cannot generate plans which include existing obligations, then our plans will not be useful for user obligation management systems. By contrast, in a partial-order planner, modifying

the algorithm to generate plans which contain the obligations and desired actions is very straightforward. A traditional partial-order planner begins with an empty plan as its starting point for searching the plan-space. By instead starting the planner with a plan which contains the obligations and desired actions, the final result will be guaranteed to contain the obligations and desired actions. Although it may be possible to restrict the search-space for GraphPlan or SatPlan, it is not clear how this can be accomplished.

It has also been shown by Nguyen and Kambhampati [36] that partial-order planners can have their efficiency improved significantly by applying techniques originally developed for GraphPlan-based and Satisfiability-based planners. They use a hybrid approach which uses a planning graph to create better heuristics which are then used in partial-order planning. As such, the performance gulf between the different types of planners is not as wide as it initially appeared.

6.2.1 Partial Order Planners

As such, we adopt partial-order planning as the means to our end. Partial-order planners, also called least-commitment planners, are plan-space search engines. As it runs, a partial-order planner keeps track of a set of candidate plans. A candidate plan has a set of actions, a set of ordering constraints, and a set of constraints on variables. It also has a list of flaws, such as unsatisfied goals or conflicts between existing actions. The unsatisfied goals may either be clauses from the end goal or from preconditions from the actions in the plan.

The planner chooses a flaw and attempts to remedy it by adding new steps, links between existing steps, ordering constraints, or variable constraints. This creates zero or more new candidate plans. If a plan has flaws which cannot be fixed, then the candidate plan is discarded and other candidate plans are examined instead. Through this process, the planner will eventually find a plan with no flaws, which is necessarily a solution to the initial problem.

A plan with no flaws will naturally be accountable, as all preconditions for actions will be met by other actions which are guaranteed to occur before them. As such, we can simply run the planner and receive an accountable plan. Or rather, we could if we could run the planner indefinitely. Because we wish our system to be responsive, we must limit the amount of time the planner spends searching for a flawless plan. In practice, planners are

generally given restrictions in terms of time or number of plans considered which bound their running time. As such, even planners which are theoretically complete are not guaranteed to find a plan within the time bounds.

We choose to use the UCPOP [37] planner, in particular, as our base for demonstrating that this technique can be effective. We choose the UCPOP planner because it is a partial-order planner which is well established, and therefore likely to be free of errors. It supports an expressive input language. And it can also be modified to support our needs.

We would have liked to use RePOP [36], which is based on UCPOP but is faster due to improved heuristics. However, the heuristics for RePOP require building a planning graph. This is not inherently problematic, except that the planning graph is extended from the based state to a point where all the goals are satisfied. The problem is that in this case, our planning problem has no explicit goals. Rather, we are sort of "gaming" the planner by giving it an initial plan which has flaws even though its final goals are trivially satisfied. As such, it is unclear how to construct a useful planning graph in our particular circumstances.

This is not to say that it would be impossible to adapt RePOP to our problem. It could likely be accomplished by developing an alternate end condition for the planning graph. However, since this is largely a proof-of-concept stage, we have elected not to tackle that problem at this time. As such, we have chosen UCPOP, which is likely to be slower than RePOP, but should otherwise be no less effective.

6.2.2 Modifications

Even with our planner chosen, solving the problem is not as easy as simply asking the planner for a solution. There are several steps which must be undertaken. The first is that the system must be translated into the input language for the planner, in this case, ADL (Action Description Language). This approach will work only for those systems whose operation can be translated into an input language which can be understood by a partial-order planner.

Fortunately, ADL is quite expressive and most reasonable access contol systems should be able to be translated into it. Actions in the access control system will become actions in the planning domain, carrying equivalent preconditions and effects. Unlike simpler planning languages, such as STRIPS, ADL can express disjunctive conditions and universally quantified conditions and effects. For example, policies which say things such

as “This file can be read if no users are currently writing it” or “DeleteAll will delete all files” can easily be expressed. It can also express conditional effects, such as “Revert(file1) will overwrite file1 unless no changes have been made”.

However, there are some restrictions to ADL. ADL is restricted to a finite set of objects and all state must be expressed through the truth or falseness of first-order logical propositions. As a result, it is difficult to create counters or other more complex representations of state. ADL has a special mechanism for counters, but counters are not well supported by the planner and their use will make it more difficult to find plans quickly. There do, therefore, exist situations for which this approach is not well suited.

But for the majority of access control languages, we should be able to translate an access control system and policy into ADL and feed it as input to the planner. However, this alone is not enough. We must also make certain modifications to the planning process in order to ensure that the plan which is output will have the properties which we desire.

Initial Plan

The first of these is that we must modify the planner to accept an initial plan. Because partial-order planners do not remove steps from plans under consideration, we can guarantee that the output plan will be a super-plan of the existing plan. That is, its set of actions, set of variable constraints, and set of ordering constraints will each be a superset of those sets from the initial plan.

Actions

Our first step, of course is to add the actions which we wish to be included in the final plan. The set of actions should be the union of the set of obligations and the set of actions which the user desires.

Along with this set of actions, we also must create a set of flaws, as we do not know how the preconditions for the actions are going to be satisfied. There may be multiple actions which could satisfy the precondtions, and we leave it up to the planner to create appropriate linkages. So, in our initial plan, all the preconditions for actions are unsatisifed, and so must be on the list of flaws.

Variable Constraints

The second step is to add an appropriate set of variable constraints. Because the obligations and desired actions must both be fully instantiated, we can simply add the existing bindings. That is to say, for neither obligations nor desired actions will there be any free variables. We will have obligations which say "Alice should send Report.txt to Bob". We will not have obligations which say "Alice should send some file to some user".

As such, when we have a "SendFile" action which takes three arguments, the user who is sending it, the file being sent, and the user to whom the file is being sent, we can know that any obligations or desired actions state the values for all arguments. In a partial-order planner, these arguments are represented by variables. This is because in an arbitrary planning domain, not all actions are going to initially have all of their arguments specified. We merely need to add bindings which specify that the variables for the action associated with a given obligation are bound to the values of the arguments for that obligation. That is, if we have the planning action "(SendFile ?U1 ?F ?U2)" and an obligation, o_1 which requires that Alice send Report.txt to Bob, then we add the binding that for action o_1, $?U1$ is bound to "Alice", $?F$ is bound to "Report.txt", and $?U2$ is bound to "Bob". No additional variable constraints are needed, so we now have both our set of actions and of variable constraints.

Timing

The more difficult problem is combining the different timing models for user obligation management systems and partial-order planners. In a user obligation management system, any obligation must have a distinct start and stop time. By contrast, in a partial-order planner, there is only a partial order which describes the relationship in time between any two actions. In order to use a partial-order planner to make plans for a user obligation management system, we must translate the discrete times into a partial order which is used to form the initial plan. Then, when the planner returns a plan, we must translate the partial order back into a set of specific time intervals to assign to the new obligations. Obviously, existing obligations should retain their time intervals unchanged.

In a partial-order planner, the partial order represents the relationship "must happen before", which we will call $\prec$. In a user obligation management system, there are

two different types of accountability which the system can attempt to maintain. In strong accountability, if one obligation is dependent on another then it is guaranteed that their time periods will not overlap. As such, there exists a "must happen before" relationship between any two obligations if and only if their time frames do not overlap. That is, for any two obligations $o_1 \prec o_2 \Leftrightarrow o_1.t_e < o_2.t_s$. For weak accountability, this is not the case because there can be overlapping obligations where the one with the earlier deadline must happen before the one with the later deadline is permitted to occur.

Note that a dependency between obligations should imply a $\prec$ relationship. If obligation o_2 depends on o_1 occuring then $o_1 \prec o_2$ should be true. However, the $\prec$ relationship does not necessarily imply a dependency. If $o_1 \prec o_2$ is true, this could either be because o_2 depends on o_1 or simply because o_1's time frame occurs entirely before o_2's time frame. In a strongly accountable system, a dependency between two obligations implies that their time frames do not overlap, so $\prec$ is easy to determine. In a weakly accountable system, both either dependency or non-overlapping time periods can cause a $\prec$ relationship.

As such, the mapping between strongly accountable systems and partial-order plans is more straight-forward. It is possible to adapt this approach to weakly accountable systems, but only when it is possible to do an analysis and determine for which overlapping pairs of obligations a "must happen before" relationship exists. The difficulty is that either allowing the planner to add inappropriate ordering constraints or disallowing the planner from adding appropriate ordering constraints will result in planning problems. Since this analysis must be exact, it is specific to the policy language in use and perhaps to the specific policies used. As such, we focus in this thesis only on failure feedback in strongly accountable systems.

Thus, let us examine how we would create the initial plan in a strongly accountable system. As we stated earlier, our set of actions in the initial plan is the union of the set of obligations and the set of desired actions. However, the obligations should not be allowed to happen in any unconstrained order. Since we are looking at a strongly accountable system, we can assume that any two obligations which are in a "must happen before" relationship must also have time frames which do not overlap.

Thus we can also determine an initial partial order to give as part of our initial plan. If two obligations do not overlap then the earlier one should be in the $\prec$ relationship with the later one. If two obligations do overlap, then they can happen in either order. As

such, they should be incomparable in our $\prec$ relationship.

Because we are attempting to find a plan in which the desired actions can happen, we do not constrain their timing at all, leaving them incomparable with all of the obligations and with each other. This is not to say that they will remain incomparable. As you recall, one of the actions of a partial-order planner is to add new ordering constraints between existing actions.

One of the properties which we proposed for the plan is that it contain the actions which the user desires. However, our property does not require that they happen in the same order which the user requested them. If such a property is desired, and it may well be in systems where the effect of a set of actions varies greatly depending on its order, we could simply add additional ordering constraints at this point. Specifically, we would simply apply the same rule to the desired actions which we applied to the obligations. For any two desired actions, da_1 and da_2, $da_1 \prec da_2$ if and only if the requested time period for da_1 happens completely before the requested time period for da_2.

Together with the set of obligations and the set of variable constraints, this set of ordering constraints defines our initial plan. However, there is one additional modification which must be made to the planner.

Timing Constraint Restriction

One of the actions which a partial-order planner often does is that it adds ordering constraints between actions. In our case this can potentially be problematic. Specifically, if there exist two obligations which overlap, then they are incomparable in $\prec$. Put another way, there is no ordering constraint between them. However, if the planner were to add an ordering constraint between them, adding either that $o_1 \prec o_2$ or $o_2 \prec o_1$ this would mean that they could no longer have the same time interval when we tried to translate the partial order back into time intervals.

For example, if we had two obligations, o_1 and o_2 with intervals $[1, 10]$ and $[5, 15]$ respectively, then if the planner added $o_1 \prec o_2$ to the plan, translating that order back into time intervals would require that $[5, 10]$ would now need to be divided between o_1 and o_2. Otherwise o_2 could happen at time 7 and o_1 at time 9, and the property of "must happen before" would not be respected and the resulting set of obligations would probably not be accountable. Likewise, if $o_2 \prec o_1$ were added, both would have to be squeezed into $[5, 10]$

without overlapping each other such that o_2 would be required to happen before o_1.

As such, we must modify the planner to guarantee that no new ordering constraints are added between obligations. This requires modifying the planner to be explicitly aware of what actions represent obligations and changing it so that it remembers what the initial set of ordering constraints over obligations is. Obviously, there is no problem if the planner attempts to add an ordering constraint which already exists.

Our modifications to the planner have to go even a step beyond this, however. In addition to not allowing any new ordering constraints over obligations, it must also not add any ordering constraints which imply new ordering constraints over obligations. The $\prec$ relation is transitive, and, as such, if there exists a new action, x, in the plan, and the ordering constraints $o_1 \prec x$ and $x \prec o_2$ are added to the plan, this is going to imply $o_1 \prec o_2$. As such, we detect this through the following algorithm:

$S \leftarrow$ the transitive closure of our initial ordering constraints

$C \leftarrow$ find the transitive closure of our current ordering constraints

$C' \leftarrow \{(a, b) \in C$ such that $a \in O \wedge b \in O\}$

Return $C' = S$

If this returns *true* then no new obligation ordering constraints have been added. If it returns $false$ then at least one has. As such, every time we evaluate whether or not a new link may be added to the plan, we run this algorithm and if it returns $false$ we disallow the link from being added. This works because the transitive closure of a set of ordering constraints contains all ordering constraints implied by the original set. Of course, for efficiency, we only need to compute S once (since it does not change) and store it for later comparisons to C'.

With this modification complete, we can now use the planner to generate plans which will be accountable under the timing model of the planner. However, we still must convert these plans back to the model of time used for obligations.

6.2.3 Converting from Planner Output to Obligations

We now need to convert the plan that the modified partial-order planner has returned into a set of obligations. Obviously, the actions in the plan that correspond to the

original obligations should simply map back to themselves. For the desired actions and any additional actions suggested by the planner, we must create new obligations. The actions in the final plan will all be fully instantiated, that is to say, they will not contain any free variables. As such, which action is involved and what are the arguments to that action are will be the same as they are in the plan. However, it is less clear what the time-frame for each obligation should be.

Generally speaking, it should be possible to give time frames to the obligations in the plan. However, there is one scenario in which this will be impossible. Because the planner only sees the partial order, it is not aware of how close together any two obligations are. As a result, it is possible that we could have a plan with the ordering constraints $o_1 \prec da_1 \prec da_2 \prec ... \prec da_{99} \prec da_{100} \prec o_2$ when the time period for o_1 is $[10, 15]$ and the time period for o_2 is $[16, 20]$. Obviously we are not going to be able to reasonably squeeze one hundred actions into a single time click.

Shortly, we are going to describe an algorithm which translates from the partial-order plan into a set of obligations. One effect of this algorithm is that it will detect when a plan cannot be successfully translated. If we wish to ensure that plans which cannot be translated are not created, we could modify the planner to run this same algorithm on candidate plans in order to eliminate time-shortage problems. However, modifying the planner in this way would substantially slow it, and, as such, we have not implemented this in our planner.

A simpler option would be to specify pairs of obligations between which new actions could not be added. In order to implement this, we would simply remove any such pairs, (o_1, o_2) from S and C in the algorithm used to check the new timing constraints. If the plan being considered contained an new action x such that $o_1 \prec x \prec o_2$ then this would imply that $o_1 \prec o_2$ would be in C', and as such, C' would not equal S. Thus, this would prevent any action from being added which is specifically scheduled to be between o_1 and o_2. This would not prevent actions from overlapping either o_1 or o_2 or the time in between them. It would only prevent actions from being scheduled to occur both after o_1 and before o_2.

In practice, we have not implemented this mechanism in our test planner because we believe that this situation will be relatively unlikely in practice. However, it would not be a difficult modification to make.

Now let us describe how to transform the partial order into a set of time frames for

obligations. What we present here is a convenient and practical algorithm which will both detect if such a conversion is impossible and create a useful set of obligations. However, it is not the only possible algorithm which could be used.

Just as there are often many different total orders which are consistent with a given partial-order, so too are there often many sets of time frames which are consistent with a given partial order. Let us consider a simple example. Suppose that there are two obligations, o_1 which has time frame $[1, 10]$ and o_2 which has time frame $[41, 50]$. If the planner then returns a plan with two new actions, x_1 and x_2 and the ordering constraints $o_1 \prec x_1 \prec x_2 \prec o_2$. Even though this is already a total order, there are still several different choices for the time periods for x_1 and x_2.

We could choose intervals $[11, 30]$ and $[31, 40]$ which would nicely divide the interval, but this may not always be appropriate. If action x_1 is that Alice should send Bob some new data and action x_2 is for Bob to submit a report about that new data, then probably more time should be allowed for action x_2. Even though the submission of the report itself happens quite quickly, the process of figuring out what to submit will likely take some time. So we might want to consider $[11, 15]$ and $[16, 40]$ as better intervals.

We could also leave some time in between the obligations in case later obligations might need to be scheduled. We could, therefore, choose $[16, 20]$ and $[31, 35]$. The difficulty is that in the end, there is no one right answer for which of these will be best. Unfortunately, domain-specific knowledge is needed to determine which approach is best, and even then there will probably be some guesswork.

Therefore, we present this algorithm as a demonstration that we can transform an arbitary partial-order plan into a set of obligation time periods, but not as the only correct way to do so.

Algorithm

To begin with, we wish to establish whether or not a transformation is possible. In order to do so, we assume that the new obligations have time periods which are as short as possible. As such, their initial time periods will have the property that $t_s = t_e$. Later in the process, we will relax this to create more manageable time periods, but we use these minimal obligation time periods as a starting point. Also, for brevity, we use $x.t$ to represent the start/end time for the new obligation associated with x, an action from the plan.

The next step is to convert the ordering constraints which exist relative to the obligations into ordering constraints which exist relative to specific points of time. That is, if we know that $o_1 \prec x_1$, this implies that $o_1.t_e < x_1.t$, and if $x_1 \prec o_1$ then $x_1.t < o_1.t_s$. As such, we now have a set of ordering constraints over integers and unknowns.

We allow this to define a directed graph in which each vertex is either an integer or an action and an edge exists between any two varticies if a less than relationship exists between them. Specifically, an edge from v_1 to v_2 exists if and only if $v_1 < v_2$.

Next we begin filling in values for $x.t$ such that each $x.t$ is as early as possible. We are going to be assigning time values to the vertices in our graph. Integer vertices are considered to have their number as their time value. The vertices which correspond to actions will initially have no time value, but will be assigned one through the course of the algorithm.

First, we do a topographic sort of the graph. This defines the order in which we visit the vertices. If we are visiting a vertex which has in-degree zero then if it is an integer, we do nothing. If it is a vertex which corresponds to an action, we assign it a time equal to the current system time plus one. When we visit a vertex, v, which has an in-degree greater than zero, we consider the set of nodes which have edges which point to v. We will call this set I. We can know that all vertices in I have been previously visited and, as such, have been assigned a time value. If v is an integer vertex, then we check I to determine whether or not there is any member of I with a value greater than or equal to v. If there is, we report failure and exit. If not, we proceed. When we visit a vertex which corresponds to an action, we find the maximum time value of all vertices in I, which we will call m. We then assign $m + 1$ to v.

Thus we have, when possible, created an initial set of time intervals which conform with the plan, and therefore the resulting set of obligations is accountable. Having completed our forward pass, we may now relax the time periods of the obligations by going back over the obligations utilizing one or more backwards passes.

In the backwards passes, we may stretch the time intervals for the new actions backwards. At this point, we are going to treat the vertices as time periods again, perhaps stretching the obligation out such that the start and end times are no longer identical. We visit the verticies in the reverse of the order that we did previously, thus ensuring that we will visit no vertex until we have visited all of the vertices to which it points. When we

visit an integer vertex, no changes are made. When we visit a time period vertex, we can increase both its start time and its end time. The only restrictions are that its start time must not exceed its end time and its end time must not exceed the start time of any vertex it points to.

There are many ways that the decision of how to relax the time period can be made. All changes which obey the restrictions outlined above will result in accountable systems. As such, this is where the possibility of system-specific choices comes into play. There are a large number of ways that the relaxation could be done, including making multiple passes over the graph. We will, however, outline one reasonable way which would be likely to produce reasonable results in many cases.

For a given time-period node, x being visited, find the integer node, i, such that $\frac{x.t-i}{pathlen(i,x)}$ is minimized where $pathlen(i,x)$ is the length of the shortest path from i to x measured in number of edges. To put it more simply, we are finding the fixed time before x such that the average time available for each obligation between them is least. Then we find the node v pointed to by x which has the lowest start time. We then relax x's time period to be $[v.t_s - \lfloor \frac{x.t-i}{pathlen(i,x)} \rfloor, v.t_s - 1]$. Note that this will always produce sensible results as $\frac{x,t-i}{pathlen(i,x)}$ is bounded below by 1. If it were less than 1, then our forward traverse would have been a failure.

This will not be an ideal algorithm for all circumstances, but in most cases, it should help ensure that obligations are given reasonably equitable time slots, while still allowing individual obligations to have longer time periods when it does not cause problems for other obligations.

Now that we have described all of the needed modifications to the planner and the steps necessary to translate obligation problems into planning problems and plans into obligations, we need to evaluate the effectiveness of our approach.

6.3 Evaluation

Having proposed the use of an artificial intelligence planning tool to generate plans for users when actions are denied, we wish to establish that this approach is, in fact, going to be an effective means of solving the problem. In order to establish this, we need to demonstrate several things. The first is that the input lanugage used by the planner can

be used to model real-world access control languages. The second is that the modified planner can solve the problem a significant percentage of the time. The third is that the modified planner can solve the problem within reasonable time. And fourth is that the plans produced are going to be relatively efficient.

The language which we have chosen to model is miniARBAC [46]. MiniARBAC is a modified subset of ARBAC[45], the Administrative Role-Based Access Control language. MiniARBAC is a language used to describe the allowable changes to roles and permissions in a Role-based Access Control (RBAC) system. In modeling miniARBAC, we are both modeling a generic RBAC system and a system by which changes in permissions can occur. Although ARBAC systems are not widely deployed, RBAC systems are in common usage and miniARBAC is a strict superset of RBAC.

There are five types of miniARBAC policies: those which control the granting of roles to users (CanAssign), those which control the revoking of roles from users (CanRevoke), those which control the granting of privileges to roles (CanAssignP), those which control the revoking of privileges from roles (CanRevokeP), and policies which specify static, mutually exclusive pairs of roles (SMER).

The first four types of polices are similar, and we will first discuss them together. Each of the first four policies takes three arguments. The first is an administrative role. The change must be invoked by a user who has that administrative role.

The second is a role expression. The role expression is a conjunction of positive or negative role membership tests. That is, for a given user or permission to satisfy a given role expression, there must be a mapping between the given user or permission and all roles which are listed in the positive, and there must be no mapping between a given user or permission and any role which is listed in the negative. For example, if the role expression was “R1, not R2” then only users who are members of R1 but not of R2 would satisfy the role expression. Also, that expression would be satisfied by a permission such that the permission was available to role R1, but not to role R2.

The third part of a rule is a set of roles. These roles are the roles which may be granted or revoked to users or to which a given permission may be granted or revoked. If a user, Alice, attempts to grant another user, Bob, a role, then it is successful if there is a policy rule such that Alice has the administrative role from the first part, Bob satisfies the role expression, and the role being granted is in the role set in the third part. Revocation

works likewise. Similarly, if a user, Alice, wants to grant a new permission to a role, then there must exist a rule such that Alice has the administrative role from the first part, the permission matches the role expression which is the second part, and the role to which the permission is being granted is in the role set which is the third part.

The fifth type of policy rules are static mutually-exclusive role (SMER) pairs. The SMER rules are simply pairs of roles such that no user is allowed to have both roles simultaneously. They take the form "SMER(role1,role2)". Note that the SMER relationship is symmetric.

6.3.1 Policy Translation

To establish that ADL can be used to model miniARBAC, we have implemented a translator which can translate arbitrary miniARBAC policies into UCPOP problem domains. It inputs policies in miniARBAC along with the current state, a list of obligations, and a list of desired actions. It outputs a problem domain and a problem in ADL, the input language of UCPOP. When this problem is solved, the result is a new plan which will be accountable, contain all of the obligations, all of the desired actions, and possibly some additional actions.

In order to do this translation, we first parse the miniARBAC policy and a file which contains additional information about the current state, the obligations, and the desired actions. Our translator takes in the standard miniARBAC policy statements, as well as statements which describe the current mapping between roles and users and between roles and permissions and statements which describe obligations and desired actions.

Specifically, we accept statements of the form "HasRole(Role,User)" and "HasRoleP(Role,Perm)" where "Role", "User", and "Perm" are replaced by the names of a role, a user, or a permission, respectively. These statements are used to describe the existing mapping. We do not check the existing mappings for conflicts with the SMER rules, instead assuming that they have been enforced.

We presume that the set of available actions consists of assigning and revoking roles from users and permissions from roles and doing actions which are allowed by the permissions assigned to the roles. Specifically, we assume that there is a one-to-one mapping between available actions and permissions. This seems consistent with normal practice in RBAC systems, however, this is merely a convenient assumption, not a necessary one. It

would be quite possible to relax this assumption at the cost of some additional complexity, but we have chosen not to do so at this point.

We also accept statements of the form "Obligation(User,Action,t_s,t_e)" where "User" is a user, "Action" is one of the available actions from the miniARBAC system (as described above), and $[t_s, t_e]$ is the time period for the obligations. The last type of statement we accept are statements of the form "DesiredAction(User,Action)" where "User" is a user and "Action" is one of the available miniARBAC actions. All actions must be fully qualified. So, if the action involves granting or revoking things, the specifics of what is to be granted to or revoked from which users or roles must be included.

With this information in hand, we can translate the policy into a ADL domain. First, we simply create a list of all roles, users, and permissions listed in the policy and current state. We assume that users, roles, and permissions which appear in neither the policies nor the current state are irrelevant. These lists will become part of the initial set of facts of the planning problem.

Next we examine the actions available. If we were to look at what are the necessary preconditions for granting a given role, we would see that it differs depending on what role one is trying to grant. As such, the different grants cannot be modeled as the same action. Thus we must separate our actions out by the roles they effect.

If we have n different roles, then we have $4n + 1$ actions. For each role we have four actions: grant role to user, revoke role from user, grant permission to role, and revoke permission from role. For example, if we had a "Manager" role, then we would have the actions "GrantRoleManager(AdminUser,User)", "RevokeRoleManager(AdminUser,User)", "GrantPermManager(AdminUser,Perm)", and "RevokePermManager(AdminUser,Perm)", where "AdminUser", "User", and "Perm" are input arguments to the actions. Specifically, the "AdminUser" is the user trying to make the change while "User" is the target of the change, that is, the user who will gain or lose the role.

Our final action is the one which lets us do actions associated with permissions granted by roles. We call it "doAction(User,Perm)". Unlike the others, it's precondition is always the same. "doAction(User,Perm)" is allowed when and only when there exists a role, r, such that "User" belongs to r and r has permission "Perm".

Now that we know what the actions are we can find their preconditions and effects. The preconditions for the role-change actions must be calculated from the policies.

Essentially, the precondition for a given role-change action is a logical or of all the different conditions from policies, since as long as any one policy matches, then the action will be allowed. For any one rule, our precondtion requires that the adminstrative user has a role which matches the administrative role from the policy and that our targeted user or permission matches the role expression. We also must ensure that if a role is being granted, that it is not granted in violation of any static mutually exclusive roles rules. As such, when we have a "GrantRole" action, we must check the SMER rules as well.

Generally, speaking the effects of the actions are more straight-forward. Grant and revoke actions change the state appropriately. DoAction has no effect on the access control state.

For example, if we had a set of three policy rules:

canAssign(Admin,Accountant AND NOT Manager, {AccountAuditor})

canAssign(Admin,StaffAuditor, {AccountAuditor})

SMER(AccountAuditor,AccountUser)

This would be translated to:

```
(:operator GrantRoleAccountAuditor
    :parameters (?A ?U)
    :preconditions (AND (OR (AND (hasRole Admin ?A) (hasRole Accountant ?U)
                                 (NOT (hasRole Manager ?U)))
                            (AND (hasRole Admin ?A) (hasRole StaffAuditor ?U)))
                        (NOT (hasRole AccountUser ?U)))
    :effect (hasRole AccountAuditor ?U))
```

6.3.2 Test Generation

In order to evaluate the effectiveness of the planner we must have a sample set of policies, obligations, and desired actions. Unfortunately, real-world policies are very difficult to come by. Many companies are understandably hesitant to share details concerning their security policies with outsiders. As a result, most of the research in this field has used hypothetical use cases to prove their relevance. Because we are attempting to establish not only that our approach can model the policies, but that it can solve the problems which arise, using use cases alone is not going to be sufficient.

Instead, we automatically generate randomized policies. These are not completely unstructured policies, but, rather, we first generate some level of organizational structure and then use that to generate policies.

Specifically, we model an organization in which roles can be divided between operational roles and administrative roles. Each administrative role has domain over some of the operational roles and also over zero or more other administrative roles. We say that one administrative role has control over another administrative role if it can determine either who can enter that role, who can leave that role, or both. The control relationship between different administrative roles forms a directed, acyclic graph. In practice, we first generate a directed, acyclic graph and then use this to generate the rules governing administrative roles.

For a user to be in an administrative role, the only requirements are that she is a member of the generic administrative role ("Admin") and that she is not currently a member of any other administrative role. We choose this policy because it is a likely real-world policy which could be used to prevent administrators from consolidating too much power.

The operational roles are divided into different levels. The lower levels contain more roles than the higher levels. Each operational role has a set number of assign rules and revoke rules. The role expression for these rules is chosen randomly, but with certain restrictions. Roles which appear in the expression are most likely to come from the same level as the role whose rule is being created. However, roles from levels above or below may also appear in the negative with a lesser probability. And roles from a lower level, but not a higher level, may appear in the positive with a similar probability.

We choose this method for generating role expressions because we want some structure to our expressions. It seems sensible that exclusionary roles could come from either above or below a given role, but it is unlikely that a low level role would require a role from a higher level in an organization.

We also must fill in the first part of the rule. In order to do this, we simply choose a random administrative role.

Our current system state is also generated in a largely random manner. We create users and assign them to the same level structure as the roles. Then users are given roles which usually match their levels, but they may have roles from above or below their level

with a lower probability. Our aim is to attempt to emulate the likely structure of a real organization. We also generate a set of administrative users which each have the base Admin role and also one other randomly chosen adminstrative role.

Next, we must generate obligations and desired actions. For our obligations, we choose actions which are available to our users in order to ensure that the state is accountable. This is a fairly basic set of obligations, but is sufficiently complex to interact with the desired actions in interesting ways. For the desired actions, we wish to create actions which would be denied. As such, we perform a time-limited reachability analysis of the generated policies to see if we can find a role for each user which they are not currently in, but which is reachable to them. We then give them a desired action which requires that role. Thus, the planner must determine how to get the user into the given role without disrupting the existing obligations. In the event that we find that there are no new roles reachable to them, then we simply give them a desired action which requires a role which they already have.

6.3.3 Results

Now that we have created a randomized system, we can vary the parameters of the system and generate test cases which we can use to validate our approach. Specifically, we can observe three things. The first thing is whether or not the planner was successful. The second thing we can measure is the amount of time it takes to generate the plan. The third thing is how many additional steps were added to the plan.

Test Set-Up

All tests were conducted on an Apple MacBook running OS X 10.5.4 with a 2.2 gigahertz Intel Core Duo 2 and 2 gigabytes of RAM. The planner ran in Gnu CLisp 2.43 and all of the code was compiled by Gnu Clisp before being executed. All times measured are processor times provided by metrics from Gnu Clisp. Processor speed scaling was disabled during the testing runs to ensure that processor times would be consistent. Measured processor time was generally proportional to wall clock time, but wall clock time was not recorded.

The test generator has a large number of degrees of freedom. Rather than varying

them all, we concentrated on the ones most likely to make a difference in the results. We will describe the quantities we varied when we describe the specific sets of tests.

However, first we will describe the quantities which were kept constant through all the tests. Obligation times ranged from 0 to 200 with obligations being from 0 to 19 time units long. Each obligation's time period and length were determined via a uniform random function.

In all cases, we had 10 administrative roles which were filled randomly by 40 administrative users and 25 edges in the directed acyclic graph which represented the relationship between different administrative roles. Each relationship had a probability of 0.1 of being a grant-only relationship, 0.1 of being a revoke-only relationship, and 0.8 of being a grant-and-revoke relationship.

We had three levels of operational roles with 5, 10, and 20 roles in each of the levels. We had 140 users divided into levels and then assigned to roles. Roles for users are assigned from the same level as the user with probability 0.8 and from levels above with probability 0.1 and below with probability 0.1. Selected roles obviously can not exceed the top or bottom level.

Each operational role had two grant rules and two revoke rules. Each rule has three clauses in its role expression. The clauses are positive with odds 0.6 and negative with odds 0.4. The roles for the clauses are drawn from the same level with a probability of 0.9. For positive clauses, roles are drawn from the level below with probability 0.1. For negative clauses, roles are drawn from the level below with probability 0.05 and from the level above also with probability 0.05. Obviously, roles are not drawn from above the top level or below the bottom level.

In each test, only a randomly selected group of users were given obligations. We wished to increase the odds that a conflict would exist between obligations and desired actions. As such, we chose a subset of users and assigned all of the obligations and desired actions to that subset.

Tests Run

We ran four sets of tests which had different purposes. For the first set of tests, we simply wished to see how well the system would handle an increasing load. In each test in the set, we selected a group of five users to have obligations and desired actions. Each user

in the group was given two desired actiosn. The number of obligations was varied, ranging over the values 1,2,4,8,16, and 32. In this set of tests, each user was given 6 roles.

In the second set of tests, we tested our assumption concerning whether or not the size of the set of obligated users was important unto itself. In these tests, we maintained a total of 100 obligations, but these obligations were divided between 1, 5, 10, 20, or 25 users. We also attempted to have 25 desired actions which would be divided amongst the users, however, as 10 and 20 do not evenly divide 25, we instead had 30 and 20 respectively for those test sets. As such, it is not an absolutely strict apples-to-apples comparison, however, it is close enough to draw meaningful conclusions. In these tests, each user was also given 6 roles.

In the third and fourth test sets, we varied the number of roles given to users. Specifically, the number of roles given to each user instead of being a fixed number is given according to a Gaussian distribution. For the third test, we set the mean to be 5 and then vary the standard distribution. For the fourth test, we fix the standard distribution at 1.5 and then vary the mean. In both tests, we have five obligated users, each of which have ten obligations and two desired actions.

Success Rate

Whether or not the planner is successful should be fairly self-explanatory. The planner is given a restriction expressed in terms of the number of plans it is allowed to examine. A planner thus restricted is not complete, and may not find solutions when such exist. When it does not find a solution in the alloted time, it reports failure. In our testing, we had 15 failures in a total of 235 trials for a success rate of 93.6%. We tend to feel that this is a fairly good success rate in the circumstances, but do not rule out the possibility that it might be improved through changing heurisitics or other means.

Speaking generally, the failures occured fairly evenly across all of the different sets of tests, although they were slightly more common on the tests which involved large plans. We examined the failures to see if there was any pattern to them which could be seen and did not identify any such pattern. It may be the case that certain patterns in the policy and requirements would induce some form of a “garden path” situation which would make it difficult for the planner to produce a correct plan efficiently. There is also some possibility that there are bugs in either the planner or the test generator which still need some ironing

out, but we observed no direct evidence of this. It is most likely that the failures represent solvable problems which the planner was unable to solve within the available time.

Our tests were run in groups of 10. When one or more failures were encountered, additional tests were run such that there would be 10 successful tests whose data could be used for timing and plan size analysis.

Time

We measure the amount of time it takes to generate a plan because we wish to demonstrate that the planner can be a practical solution to our problem. If it were to take hours and hours to generate a plan, it would be quite likely that the state of the system would have changed so much in the mean time that the plan would be useless. Also, because this is for a system intended to interact with users, excessive times would be inconvenient to the intended users. That said, our expectation is that an amount of time ranging from several seconds to perhaps as long as two or three minutes would be a reasonable amount of time to wait for the planner to come back with an answer. Obviously, in systems where the state changes quite frequently, the longer end of this scale might be problematic, but we believe that in many systems, a wait of a couple of minutes will be acceptable. Our evaluation of time is fairly straight-forward.

First we examine our time results from the first set of tests. The results for this test are in table 6.1. As expected, the amount of time required grows as the number of total actions increases. We found that in some test sets, there would be a single test which would take significantly longer than the others, thus having a substantial impact on the mean. As such, we have provided both the mean and the median, as the median is not effected by single outliers.

The amount of time grows at a more than linear rate. For small amount of obligations, the time is quite small. For moderate amounts of obligations, the time is larger, but still within the acceptable range. Clearly, by the time we have 170 total actions, it is starting to stretch the limits of our acceptable time bounds, being about two and half minutes. Not included in the table is that we did attempt a single run with 64 obligations per user for a total of 330 actions. This completed successfully in 1605 seconds, which is to say roughly 27 minutes. This will probably be outside of the acceptable time bounds for most systems.

Table 6.1: Increasing Obligations

Obligations Per User	Total Actions	Mean Time(s)	Median Time(s)
1	15	4.36	1.16
2	20	1.19	1.00
4	30	2.46	1.80
8	50	6.87	5.56
16	90	24.24	23.10
32	170	152.81	148.96

In order to illustrate the effects of the growth of the number of actions on the time, we have included graph (figure 6.1) of number of actions versus time. This graph includes the data from all four test sets and not just the first one. Clearly, it shows that the number of actions is the most dominant factor in terms of the amount of time needed and that the curve of the growth is unfortunately greater than linear. This is what we expected.

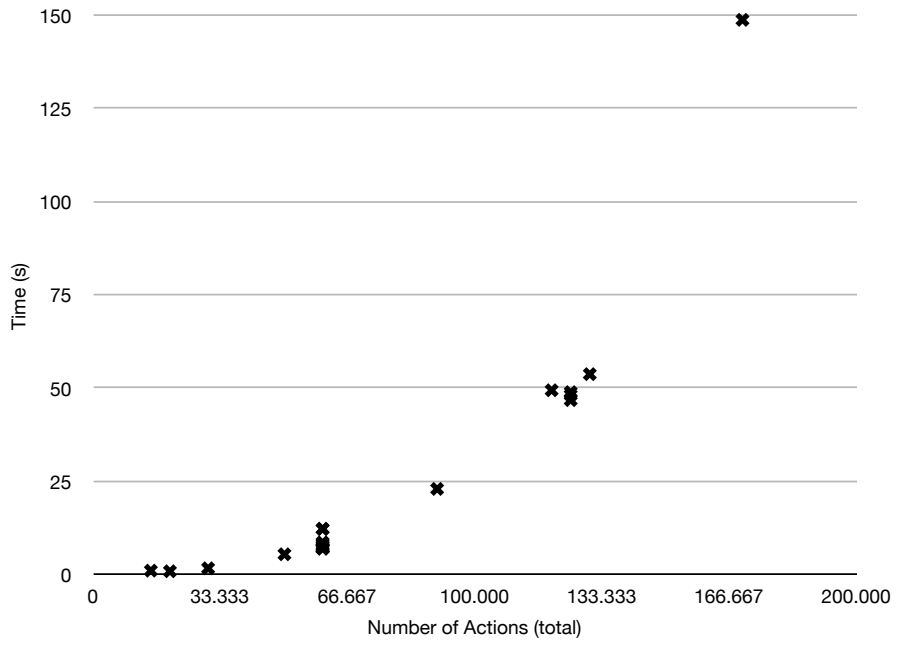

Figure 6.1: Total number of actions versus Time for all Test Sets

In the second test, we varied the number of users, while keeping the total number

Table 6.2: Action Distribution

Number of Users	Total Actions	Mean Time(s)	Median Time(s)
1	125	47.22	46.92
5	125	47.99	48.29
10	130	52.03	53.90
20	120	55.99	49.57
25	125	62.75	49.06

of obligations and desired actions the same. Our expectation is that doing so would have little effect, and our experiments bore this out. As you can see in table 6.2 and figure 6.2, there was a gentle rising curve in the mean as the number of users with obligations increased. We had actually hypothesized a gentle falling curve rather than a rising one, assuming that more conflicts would occur in the earlier cases, leading to longer times when there were less users, but this was not borne out by our tests.

It should be noted that the curve of the mean and the median diverge in the second half of the tests. We believe that the mean, in this case, reflects the influence of outliers and that the median is a more accurate reflection of expected performance. The median also more closely tracks the total number of actions, which is higher for the ten user case (130) than the other cases (120 or 125). As such, the median of the results supports the idea that the distribution of obligations over users is much less significant to the running time than the total number of actions.

For our third test, we examine how the distribution of roles effects the time required. Our hypothesis was that this would have very little effect, and this is, in fact, what the data concludes. We show the data in table 6.3 and figure 6.3. The median stays more or less level while the mean fluctuates in a manner which is uncorrelated with the distribution of roles.

The fluctuation in the mean is caused by a small number of tests which took much longer than average. For example, in the set of tests for which the standard deviation was 1.0, there was a single test which took 68 seconds, almost as long as all of the others combined. When these outliers are disregarded, the mean is similar to the median and shows only very small fluctuations. We can definitely conclude that whether the roles are evenly or unevenly distributed to the users has no significant effect on the amount of time

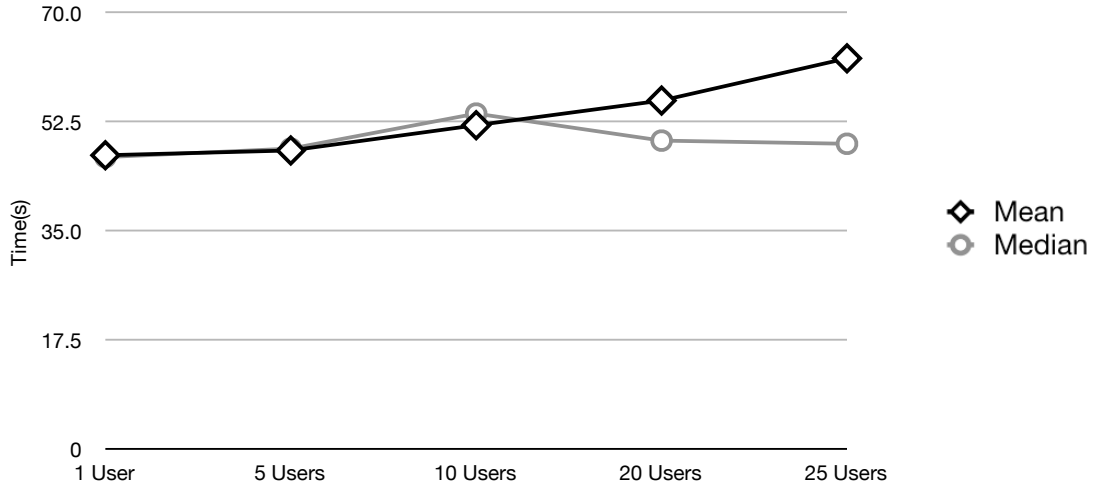

Figure 6.2: Number of Users with Obligations versus Time

Table 6.3: Distribution of Roles

Standard Deviation	Mean Time(s)	Median Time(s)
0.0	10.75	8.75
0.5	6.72	7.10
1.0	13.07	7.21
2.0	10.76	7.81
4.0	8.07	7.94

that the planner will take.

For our final set of tests, we examined the impact that the average number of roles would have on the amount of time required to solve a problem. Our results are shown in table 6.4 and in figure 6.4. In this case, there is a small but steady rising trend.

This makes sense because as the average number of roles per user increases, so does the user's likely ability to satisfy the preconditions of polices. This is not uniformly the case since policies can have negative clauses as well as positive ones. However, generally speaking, more roles will mean more preconditions which will be satisfied. As a result, the number of different options available to the planner increases, thus increasing the branching factor which usually leads to an increase in planning time.

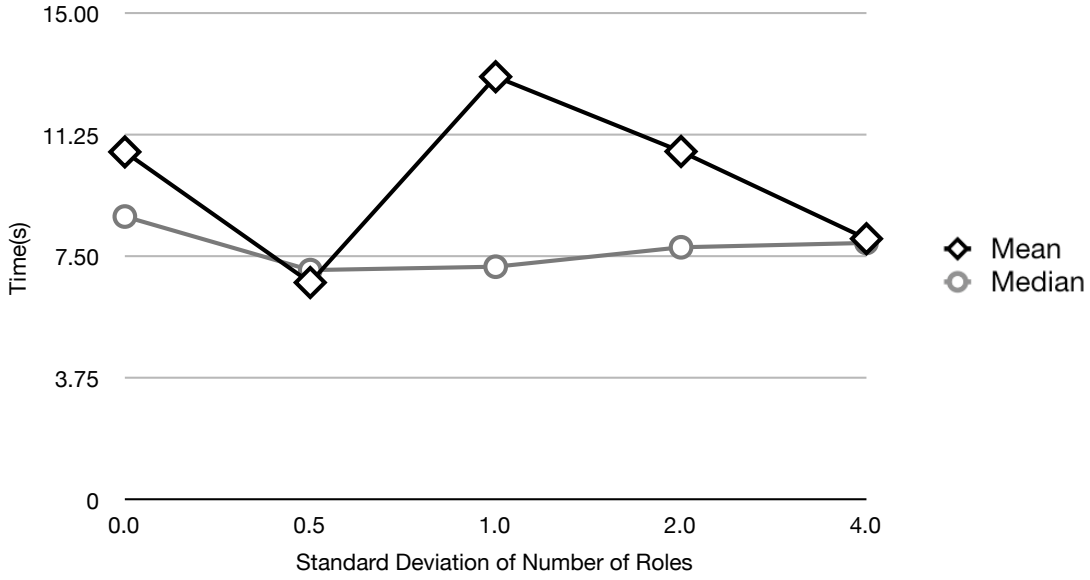

Figure 6.3: Distribution of Roles versus Time

In this particular case, it can be seen that an increase in the number of roles also leads to more outliers, more extreme outliers, and generally less predictability in terms of the amount of time the planner will take. One good measure of this is the standard deviation, which we graph in figure 6.5. The rising trend is clear. As such, we can conclude that system qualities, such as the number of roles per user, which increase the branching factor of the search are also likely to impact the search time, although not quite as severely as the total number of actions.

Number of Steps

We also measure the number of additional steps added to the plan. At minimum, the plan will have a number of steps equal to the sum of the obligations and the desired actions. Often, desired actions will require additional steps be added in order to satisfy them. However, this will not always been needed.

Our test case generation algorithm selects a random role and then uses reachability analysis to find a new role which can be reached from the start state and selects that role as being one which the user must achieve. However, due to the constraints of the access control

Table 6.4: Number of Roles

Roles Per User (mean)	Mean Time(s)	Median Time(s)	Standard Deviation of Time
1.0	7.00	7.08	0.90
2.0	7.57	7.44	1.10
4.0	7.16	7.20	1.61
8.0	9.00	8.44	3.15
12.0	17.13	12.32	16.18
16.0	53.66	12.43	116.06

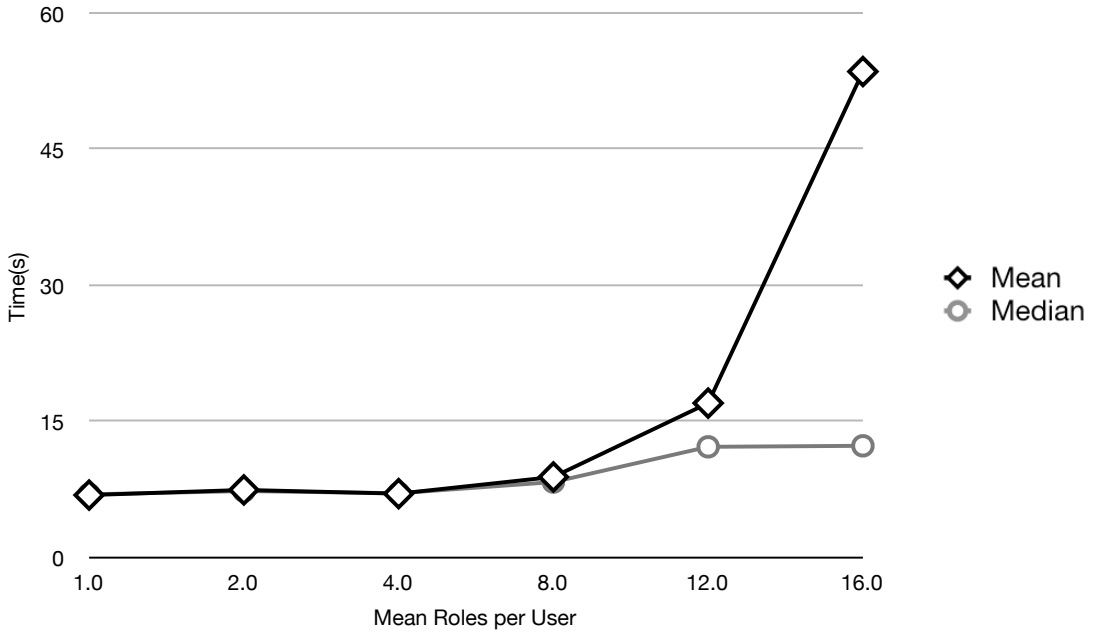

Figure 6.4: Average Number of Roles versus Time

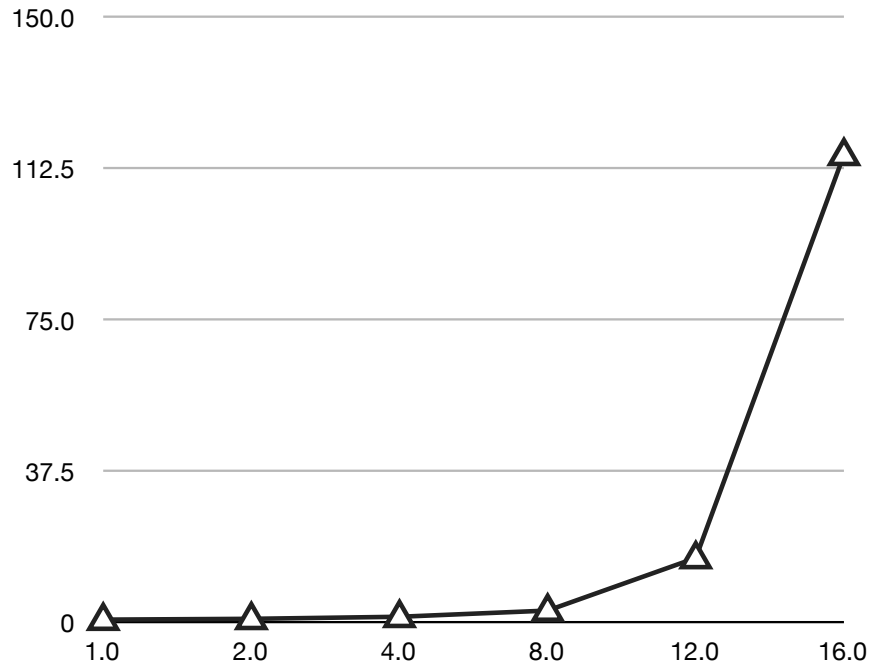

Figure 6.5: Average Number of Roles versus Standard Deviation of Time

rules, not all users have a set of roles which allows them to acquire new roles. As such, when new roles are unavailable, the system simply chooses a role which the user already has and asks them to carry out an action which requires that role. When this occurs, no additional steps are needed.

As such, the number of steps needed for a desired action varies from as few as zero to as many as seven or eight. Unfortunately, for any given test policy we are not sure what the optimal possible plan is. So instead we simply concentrate on demonstrating that the plans generated do not contain a large degree of excess and unnecessary actions. Plans which do have a large number of unneeded actions could be problematic, but it will likely not be a problem if a few plans differ fairly slightly from the optimal possible plan.

Rather than examining the raw number of steps added, we examine the number of steps added divided by the number of desired actions which were present in the problem. Looking at the different test sets, we saw no clear trends either within or across any test set. As such, we have only calculated the average number of steps added per test set: 0.33. This number indicates two things to us.

The first is that a great number of desired actions did not require any additional steps be added. This indicates that the number of new roles available to users was small on average. The second is that the planner clearly did not add any substantial number of unnecessary steps in the plans. This is not an exact quantification of optimality, but it does suggest that the plans generated are likely to be quite efficient.

As such, given both the positive time results and the fact that the plans produced are likely to be usable, we can conclude that least commitment planners are a promising tool for solving this problem for at least moderately-sized systems.

6.4 Related Work

Unforunately, there is not a very large volume of related work. Although recently, there have been more papers which emphasize attempting to combine security with usability, most of the work has come at this from a different angle than our paper. Namely, most papers look at increasing the usability of the software which provides the users with data security. The papers examine reasons that users might make bad security decisions, use programs in an insecure manner, or avoid using security programs. They then attempt to

improve the usability of these systems or propose usability guidelines.

For example, Zurko *et al.*[55] describe a system called Adage which handles RBAC authorizations. One way in which they distinguish Adage is that they intentionally designed it to be friendly to the user and ran usability studies to evaluated the ease of use of Adage. However, in this case, the user of Adage is the person making the security decisions, which is to say the administrator, rather than an ordinary user.

Most papers about usable security focus on the person making the security decisions. In many cases, this is an administrator, but in some papers, they examine the usability of programs which are run by ordinary users. One such is Whitten and Tygar [53] which analyzes the user interface for PGP 5.0. Still, this type of approach focusses on aiding the user in making appropriate security decisions.

We, instead, focus on attempting to ensure usability when the security policy is preventing the user from doing what they would otherwise like to. We attempt to preserve usability when the system is making the security decision for the user. The only other paper which we could find which takes a similar tack is the Kapadia and Sampemane paper [27] which we have previously discussed in section 6.1.

One paper which looks some at both perspectives is Besnard and Arief [4] who examine reason why users might use systems in an insecure manner. They cover both why users might make poor security decisions and why users might intentionally go around security systems in order to get work done. This is relevant to our motivation because if the user obligation management system is too difficult to use, users will find ways to avoid using it thus negating any security gains. However, they are focussed on understanding user behavior and creating guidelines for effective programming rather than any particular system.

Chapter 7

Secure Planning

In this chapter we wish to again look at systems in which the users can schedule actions ahead of time. In particular, we wish to examine the problem of what information is leaked by allowing or denying future actions. We examine whether or not allowing or denying these actions can lead to the leakage of sensitive information about future actions.

7.1 Motivation

Actions scheduled in the future are still user actions, and, as such, will need to be governed by security policies. One goal which we would like a system to achieve is that if an action is scheduled to occur at a certain time, then the action should be permitted to occur at that time. That is, the schedule of actions should be consistent with the security policies of the system. If this is not the case, then either security or reliability will necessarily suffer.

When a user attempts to schedule an action for some future time, this should only be allowed if the action will be allowed at the time it is being scheduled. However, a scheduling system which operates in this manner runs into a potential problem with information leakage. Specifically, the different actions which can be conducted in a system may have differing sensitivities. And as such, it may be inappropriate for a user to know of other actions which are scheduled to occur in the future.

Because actions are governed by policy rules, the success or failure of attempts to schedule actions may reveal information about the future access control state of the system. The access control state may not, unto itself, be sensitive, but changes to the access control

state over time will likely reveal which actions have occurred. As such, a user may be able to infer information about which other actions are scheduled to occur based on the acceptance or rejection of her attempts to schedule actions. This means that there is a potential channel that may allow users to gain information about future events about which they are may not be authorized to know.

This potential leakage of information in security-aware scheduling systems is not something which has been investigated elsewhere. In this chapter, we begin by formally modeling the workings of a scheduling system which allows users to schedule future actions, restricts the scheduling of actions to only those actions which will comply with the security policies of the system, and supports different actions having different levels of secrecy associated with them. Once we have described such a system, we then define and analyze the problem of users gaining unauthorized information about future events.

As in the previous chapter, we are again looking at a limited version of an obligation system. As it happens, the problem of information leakage is very difficult. For the time being, we have limited ourselves to a simpler system without some of the complexities inherent in obligations. We do this in order to attempt to get to the root of the problem. Specifically, the system which we will outline differs from obligation systems in that all actions are scheduled for a particular time and all actions are scheduled by users. Even with this much more limited framework, it is quite difficult to prevent information leakage.

In particular, we focus on future events because concealing actions which are presently happening is much more difficult and would require either more drastic measures or, in some cases, measures outside of the control of the security system. Although the scheduling of future actions is something which takes place primarily in the computer system, the actions themselves may have real world effects, and in many practical environments, there are other channels which could allow users to be aware of things which have happened or are happening.

For example, a company might make a decision to lay off an employee next month or to restate its earning from the previous quarter. It would not want the employee to know that he is being fired in advance of his scheduled notice, and it would not want its decision to restate its earnings known until a public announcment was made. As such, the presence of these actions on the schedule should be privileged information. However, once the event has occurred, it is essentially impossible to conceal. Despite this, we still wish to prevent

them from being inferred ahead of time.

There may also be events which will not necessarily become publically known but for which advance knowledge could be problematic. For example, if an employee were to schedule a meeting with someone in the HR department to discuss the fact that her boss is sexually harrassing her, if her boss were to infer this meeting in advance, he could perhaps apply pressure to prevent her from coming forward. A scheduling system could potentially leak this information if her boss attempted to schedule a meeting at a certain time and discovered that she would be unavailable at a time when he knew her to have no publically disclosed meetings.

Also, users may attempt actions as a means to probe the current access control state. The inferences that the user can make from the results of these attempts mean that users will always have access to some knowledge about the current state. As such, we assume that information about the current state is likely to be known by an attacker, but we do not depend on the current state being explicitly public in our definitions or approaches. Our focus, however, is on how to prevent action scheduling systems that maintain consistency with security policies from leaking information about planned actions.

One possible approach to preventing this information leakage is to enforce a total separation between the actions which a user can do and the actions to be kept secret from the user. In many practical systems, however, this is impossible. In the example of the user who is to be fired, this would require that all users who can be fired not be able to schedule any actions which have employment as a precondition. This is quite a stringent requirement and will strongly limit the usefulness of the scheduling system.

In other systems, there might be resources which are being used for different projects. If some of the projects are secret, then reservations they would make for resources would be considered secret. A complete separation of actions would require that the secret projects and the non-secret projects would have to have a separation of resources, either having duplicate resources or having them only available to the different groups at different fixed times. Any such solution is going to reduce the utility of the resources or increase their cost. In many cases, this may be a reasonable expense given the cost of information leakage. But, in others, this complete separation would be impractical or excessively expensive. As such, in this paper we propose methods for preventing information leakage without requiring complete separation of actions.

Specifically, we will first discuss techniques for generating *cover stories* in our scheduling system. In espionage, a cover story is a false story which is consistent with the facts which an outsider would be able to observe. For instance, if an agent were spying on a government installation in the forest using binoculars, he might have a cover story that he is bird watching and carry a birding book with him to reenforce his cover story. Previous research [50, 23, 24] has examined the use of cover stories in databases, looking at techniques wherein certain values or tuples appear differently to users of different security classes while maintaining consistency relative to any one particular user. This is called polyinstantiation.

For our cover stories, we propose means by which the scheduling system allows users to believe that certain actions have been added to the schedule when, in reality, they have not. Much as a polyinstantiated database should be consistent from the perspective of any one user, the schedule should also appear to be consistent from the perspective of any one user. However, the specifics of what is meant by consistency and how it is achieved is quite different between schedules and databases. We will go into more details about how cover stories work in our system, when they can be used, and what the ramifications of their use are in section 7.3.

Our second approach involves dealing with potential information leaks through cancelation and rescheduling of higher secrecy actions. This technique is quite effective at preventing leaks, but will usually have negative impacts on the effectiveness of a scheduling system by other measures. We explore the details of how rescheduling and cancelation would work, and we analyze in what situations this will be feasible and in what situations it will be problematic in section 7.4.

7.2 System Model

Before we describe and model information leakage in a scheduling system, we must first formally model an abstract system in which there are security policies and in which users can schedule actions to occur in the future. This model is, roughly speaking, a subset of the obligation system model which we used previously, but substantially simplified. We model time in our system using real numbers. However, the time model is not important, and we would expect real world systems to use clock ticks. As such, integers could be used

instead.

Initially we describe the behavior of the system without scheduling. We define a basic system as a tuple $(\Sigma, U, A, \sigma_0, t_0)$ where:

- Σ is the set of possible access control states of the system. In order to keep our model as general as possible, we do not define the type of the access control state in our abstract system. It should, however, encapsulate all information relevant to describing when an action is allowed to occur and also when it is possible for that action to occur. This goes a little beyond what is normally considered access control state since it also encompasses information about resource availability and other information which would affect the ability of users to carry out actions.

- U is a finite set of users.

- A is a finite set of actions. Each $a \in A$ has a name, a precondition, and an effect. $a = (a.n, a.c, a.e)$. The name can be any unique identifier. Each condition, c is a boolean function of the access control state, σ, and the user and the current time which indicates whether or not it is possible and acceptable for the user to take action a at the given time. If the system is in some state σ and $a.c(u, t, \sigma) = true$ then it should be possible and acceptable for a to be carried out by u at this time. If $a.c(u, t, \sigma) = false$ then it should either be impossible or forbidden for σ to occur. Note that this incorporates all aspects of what is necessary for a user to be able to take an action and not only whether the access control policy accepts the action. Each effect, e is a function which maps from $U \times \mathbb{R} \times \Sigma$ to Σ and thus indicates the change in access control state which will occur should action a be done by a given user at a given time.

- σ_0 is the initial access control state of the system.

- t_0 is the initial time for the system.

Next we describe how the system manages the schedule of future events. In addition to the set of actions which are part of the main functioning of the system, users are also allowed to attempt to schedule sets of future actions. The act of attempting to schedule a future action is not an action in A, and as such will not modify the access control state.

Also, this prevents users from scheduling a future attempt to schedule an action and other such silliness.

Rather than being governed by the normal access control policy and its preconditions and effects, we have a special policy function, P, which describes which users can schedule what actions when. Formally, we define the set of possible schedules and its policy function thus:

- $\mathcal{S}$ is the set of possible schedules. We define it to be $\mathscr{F}\mathscr{P}(A \times U \times \mathbb{R})$ where $\mathscr{F}\mathscr{P}(X)$ denotes the set of finite subsets of X. Any particular schedule S is a member of $\mathcal{S}$, which is to say that any schedule is a set of triples of a user, an action, and a time at which that user will perform that action.

- P is the schedule policy function. P is a function which decides whether or not a given user has permission to schedule a given set of actions at given times. P is in $U \times \mathbb{R} \times \Sigma \times \mathcal{S} \times \mathcal{S} \rightarrow \{true, false\}$. That is, that P is a function which takes in a user who is attempting to schedule some actions, the current time, the current system state, the current schedule, and the proposed additions to the schedule and returns whether or not this should be allowed.

At any particular time, the state of the system is represented by a triple (t, σ, S) where t is a real number representing the current time, σ is the access control state, and S is the schedule of future actions. S subset of $\mathcal{S}$ such that for all $s \in S, s = (t_s, u_s, a_s), t_s > t$. That is, all actions on the schedule must be scheduled to occur after the current time.

For the sake of simplicity, we assume that actions occur instantaneously, taking no time to occur, and that they are atomic; their executions do not overlap. In practical systems, only the atomicity is needed. The assumption of instantaneous action is only included to simplify the discussion of what state the system is in at any given time. We also assume that any action on the schedule will occur at its scheduled time. This allows us to project an expected future system state for a given time by using the current access control state and schedule. We represent this future state as $\sigma(S, t, t', \sigma_{t'})$, that is, the state which is reached when we begin in state $\sigma_{t'}$ at time t' and follow schedule S until time t. In our definition, we do not assume that all events in S occur after t' or before t or that $t > t'$.

Formally, we define $\sigma(S, t, t', \sigma_{t'})$ by assuming that any schedule $S = \{s_1, s_2, ..., s_n\}$ is totally ordered by $t_{s_k} < t_{s_{k+1}}$ for all $k \in \{1, ..., |S| - 1\}$. Next we find the greatest

$i \in \{1, ..., |S|\}$ such that $t_{s_i} < t$. If no such i exists, then $\sigma(S, t, t', \sigma_{t'}) = \sigma_{t'}$. Elsewise we continue and also find the least $j \in \{1, ..., |S|\}$ such that $t_{s_j} \geq t'$. If no such j exists, then $\sigma(S, t, t', \sigma_{t'}) = \sigma_{t'}$. When both i and j exist, we define $\sigma(S, t, t', \sigma_{t'}) = a_{s_i}.e(u_{s_i}.t_{s_i}, a_{s_{i-1}}.e(u_{s_{i-1}}, t_{s_{i-1}}, a_{s_{i-2}}.e(\ldots a_{s_{j+1}}.e(u_{s_{j+1}}, t_{s_{j+1}}, a_{s_j}.e(u_{s_j}, t_{s_j}, \sigma_{t'}))...)))$. We note that if $i < j$ then no events occur both after t' and before t. In which case, our definition yields $\sigma(S, t, \sigma_{t'}, t') = \sigma_{t'}$, which is what we would expect.

We can use this to then express the idea of a schedule being *consistent.*

Definition 10 *If, at time t, when the system is in state σ, it has a schedule S, S is consistent if it is the case that for all $s \in S$, $a_s.c(u_s, t_s, \sigma(S, t_s, t, \sigma)) = true$. That is, the precondition of every action in the schedule is going to be true at the time it is scheduled to happen. Likewise, a proposed set of actions S' is consistent with a schedule S at time t in state σ if $S \bigcup S'$ is consistent.*

Now that we have the formal definitions of behavior out of the way, let us note some features of our system which are somewhat unusual. The first is that our idea of an action is a little different from most people's intuitive ideas of what an action is. Specifically, an action is something which either depends on and/or changes the access control state. As a result, some things which we might think of as a single action, such as using a shared resource like a supercomputer, actually have to be broken down into two actions. Rather than having a single action "use supercomputer" for which we can make a reservation, we have two actions "begin using supercomputer" and "stop using supercomputer". This is necessary because whether or not anyone else can use the supercomputer changes when we begin using it. As such, this change in access control state needs to be encapsulated into an action. Likewise the later change in state when the use ends also needs to be an action.

This formalism is equivalent to the more common idea of an action in terms of expressiveness, but it does have the downside that it could make writing policies more difficult. For this reason, we allow our scheduling policy to consider sets of actions requested at the same time and to consider the existing schedule. This approach, for example, allows the scheduling policy function to impose time limits on supercomputer use, which would be impossible if we only considered actions in isolation. If actions were only considered by themselves, we would not know whether or not to grant the request to begin using the supercomputer. Once we know that the user has plans to start using it at one time and to

stop using it at another, we can consider whether or not this is acceptable use.

The scheduling policy is a function which can be set by the administrator of the system and expresses the policies governing under what circumstances a user is permitted to schedule actions. However, it is assumed that the system also enforces the requirement that schedules be consistent. Actions will only be allowed to be scheduled if the policy function permits it *and* the resultant schedule is consistent. If either of these does not permit it then the actions will not be added to the schedule.

7.2.1 Information Leakage

We have described the workings of the scheduling system, but we have not discussed information leakage. If there are certain events that we want to keep secret, it is important that the scheduling system not leak information about them. For example, if today is Monday and Bob is going to be fired on Wednesday, then he should not be able to discover this. If he attempts to schedule several different actions for Friday and they are all rejected, he may be able to reason that something unusual is happening. Depending on the details of the system state and the access control policy, Bob may even be able to look at the actions he has chosen to schedule and know that the only way that they would all be rejected is if we were not going to be an employee on Friday. This is the sort of information leakage that we are trying to define such that we can address how to avoid it.

First let us address what the boundaries are across which information should not leak. We assume there is a set of security labels, L, which form a lattice. Let us assign a security label to each user and to each action on the current schedule. We define a function $\hat{L}$ which given any user, u, or scheduled action, s, returns their security label. Formally, $\hat{L}$ is a mapping from $U \cup S$ to L. We extend $\hat{L}$ to be a total function over $U \cup (\mathbb{R} \times U \times A)$ by defining $\hat{L}(s)$ to be the least member of L for each $s \in \mathbb{R} \times U \times A$ not in S. A user's security label represents the level of secret information which she is allowed to know. A scheduled action's security label represents its desired level of secrecy. A user is only allowed to know about an action if the user's label dominates the scheduled action's label.

When a new action is scheduled, it is assigned a label. A user may assign to a scheduled action any label which their label dominates. This represents the common workplace scenario in which managers can schedule either actions which their employees are aware of or ones which their employees are unaware of.

This does not follow the Bell-LaPadula model in that we allow users to schedule actions and give those scheduled actions lower secrecy than the user's label. The reason that we chose to do this is that we would like to provide security, where possible, without complete separation of resources. Unfortunately, using a strict BLP model would require that we completely separate all actions available to different labels. In the BLP model, only reading downwards and writing upwards are allowed; reading upwards and writing downwards are not allowed.

When viewed from a mandatory access control perspective, attempting to schedule events is a form of both "reading" and "writing" in that users can infer information about the future system state and hence what actions are scheduled based on whether or not their action is accepted, and they also potentially make changes to the schedule of actions. Because of this, in a full MAC system, users would only be able to schedule actions which are visible to others who are in precisely the same security class. Doing otherwise would allow for a covert channel by which two users could attempt to schedule actions which use the same resources at the same time.

As such, we do not attempt to enforce mandatory access control restrictions to prevent information flow. Instead we allow users to effectively both read down and write down, but not read up or write up. Although there may be some circumstances in which it is valuable to prevent users from scheduling lower security actions, this is not the typical scenario in most organizations and would probably be best dealt with through complete separation of resources into different classess.

The security goal of the system is to maintain the secrecy of actions on the schedule. That is to say, actions on the schedule should only be known to users who have security classes which dominate the secrecy class of the scheduled action. We believe that in most systems it will be very difficult to prevent users from knowing the current state of the system or the system's policies. As such, actions which have actually occurred are not considered to be secret. In this sense, we are focussing on the secrecy of plans rather than actions. Keeping the system state and the actions which create that state secret is again a goal best served by complete separation of resources into different classes or levels. We wish to explore the security which can be achieved without requiring the duplication of resources or similar costs which would be required to completely separate resources into differing security levels.

In order to formally capture our notion of security, we must now introduce a model for the opponent. Formally, we model the opponent as an interactive Turing machine. Our opponent, O, is allowed to control multiple subjects.

Our opponent's goal is to use its subjects to gain information about scheduled actions which have a secrecy class which none of its subjects' security labels dominate. As such, we present to our opponent two systems, X_1 and X_2, with which it can communicate. These systems are identical in terms of current state, available actions, and policies. They are also identical in terms of the scheduled events which our opponent is allowed to see. However, the two systems differ in terms of one or more scheduled actions which have a secrecy level that none of our opponent's subjects' labels dominate. Each system has a set of secret scheduled actions, Z_1 and Z_2 such that $Z_1 \neq Z_2$. Either Z_1 or Z_2 may be empty, but obviously not both. Our opponent is told the values of Z_1 and Z_2 but not which is associated with which system. The two systems and their secret scheduled actions are presented to the opponent in random order. The total input for the machine is the set of allowable actions (A), the scheduling policies (P), the current access control state (σ), the current system time (t), the communication interfaces for the two systems (X_1,X_2) in random order, and the sets of secret scheduled actions (Z_1, Z_2) in random order. Our opponent is allowed to attempt to make any reservations which it wishes to on either system and to attempt to take any actions. It can continue doing this until the first time at which the system state will differ between the two systems (we call this the *point of differentiation*). Prior to that point, the opponent is required to guess which system has which set of secret events.

That point can be identified by examining Z_1 and Z_2. Specifically, it is the smallest time found in any triple in $Z_1 \bigcup Z_2 - (Z_1 \bigcap Z_2)$. That is, the earliest scheduled time for any upcoming event which is in either Z_1 or Z_2, but not both.

If the opponent is unable to guess which system has which schedule with probability greater than one half then the two systems are said to be indistinguishable to that opponent.

A scheme is completely secure against inferences if there exists no opponent which can distinguish between any pair of systems which meet the criteria of the trial when the scheme is used. That is, they share the same state and the same set of events known to the opponent.

Theorem 6 *If the communication a user would experience with two systems is identical until the point of differentiation, then those two systems are indistinguishable to all possible*

opponents.

Proof 3 *Let us be given two systems X_1 and X_2 which produce identical output communication given the same input communication. Let our opponent be a deterministic interactive Turing machine which we will call O.* [1] *Given some fixed input and access to X_1 and X_2, since all components are acting deterministically, this must produce some guess g which is a bijective mapping from $\{Z_1, Z_2\}$ to $\{X_1, X_2\}$. Because the opponent is a deterministic Turing machine, its output will be uniquely determined by the input. X_1, X_2, Z_1, and Z_2 must be presented to the Turing machine in some order. Potentially, this ordering could affect the output. As such, we randomize the ordering of the two systems and two sets of secret events. There are four possible equally likely orderings of X_1,X_2,Z_1, and Z_2. As such, generally, there would be four possible guesses which an opponent could produce. We will identify these by g_{X_i,Z_j} where i indicates which system, X, was presented first and j indicates which secret action set, s, was presented first.*

$Pr(g_{X_i,Z_j}(Z_1) = X_1) = \frac{1}{4} \cdot Pr(g_{X_1,Z_1}(Z_1) = X_1) + \frac{1}{4} \cdot Pr(g_{X_1,Z_2}(Z_1) = X_1) + \frac{1}{4} \cdot Pr(g_{X_2,Z_1}(Z_1) = X_1) + \frac{1}{4} \cdot Pr(g_{X_2,Z_2}(Z_1) = X_1)$. *Because X_1 and X_2 produce identical communication, it follows that $g_{X_1,Z_1} = g_{X_2,Z_1}$ and $g_{X_1,Z_2} = g_{X_2,Z_2}$. Therefore $Pr(g_{X_1,Z_j}(Z_1) = X_1) + Pr(g_{X_2,Z_j} = X_1) = 1$. Hence, by substitution, $Pr(g_{X_i,Z_j}(Z_1) = X_1) = \frac{1}{2}$.*

7.3 Cover Stories

In our system, it is considered sensitive to have certain actions on the schedule. If the existence of those actions can be inferred, then a security violation has occurred. It is relatively simple to prevent direct leakage of information, but indirect leakage is more complex. Actions have both preconditions and effects and can have complex inter-relations with other actions.

Problems will arise if users see a set of actions which are inconsistent. If they can see actions whose preconditions are not met then they will know that there must be other actions which they cannot see. Further, if they attempt to schedule an action whose precondition appears that it would be met, but the system denies their scheduling request

[1] We do not lose any expressiveness by using a deterministic Turing machine as deterministic and probabalistic Turing machines are computationally equivalent. However, the use of a deterministic Turing machine simplifies the proof.

then they know that some scheduled actions invisible to them are preventing their request. Either of these scenarios would result in information leakage and allow inferences about the existence of secret actions.

If, however, a schedule appears to be consistent, then the user will be unable to make any inferences. What we wish to propose is that the low level user always see a consistent schedule even if doing so requires making that scheduled inaccurate. To be clear, we do not wish unchecked inaccuracy to occur. The reliability of a system is also important. But if we can provide low level users with an appropriate cover story which appears to be consistent then they should be unable to distinguish whether or not high level events are occurring.

7.3.1 The Problem

In particular, we wish to focus on what happens when a low level user attempts to schedule an action whose preconditions will not be met because a higher secrecy action interferes with them. For instance, let us say that Alice attempts to make a reservation to use the supercomputer on Friday. Alice routinely uses the supercomputer as part of her job as an astrophysicist, and she knows that she has the needed permissions to satisfy the security policy. However, what Alice does not know is that her bosses have become displeased with her, and they have scheduled for her to be fired on Wednesday.

This presents a dilemma to a scheduling system. If the system accepts the action then it will have accepted a reservation for an action which will not be carried out. But if it tells Alice that she cannot make the reservation then she will be aware that something secret is happening.

A naive approach would be to have the system turn Alice's scheduling attempt down with the hope that she might believe that someone else has already reserved it for that time. Unfortunately this ruse is unlikely to work since if Alice cannot get a reservation at the time she wants she will likely try to reserve it a little later. When that time is also rejected, she would try a little later still. And this would continue until Alice realized that no matter what time she attempted to make a reservation for, it was rejected.

A second possible approach might be to simply not tell Alice whether or not her request was accepted by the system, but this risks making the system quite difficult to use. Beyond that, this is also not likely to work because users will soon realize that a lack of

response is just a "secret" form of rejection. So, the system cannot accept her reservation, it cannot reject it, and it cannot simply remain silent.

7.3.2 Our Approach

Instead we suggest that the system should appear to accept it, but not accept it in actuality. That is, the system should maintain a cover story, a schedule of events visible to Alice and other users at the same security level which contains scheduled actions which will not actually occur because they will be preempted by secret actions.

This idea is similar to the polyinstantiation idea which has been described for multilevel databases [50, 23, 24]. We wish for users in a particular security level to both see only scheduled actions which they are privileged to see but also to see a consistent schedule. In order to do this, when users attempt to schedule actions which are consistent with the schedule which they can see, we allow these attempts to appear to succeed.

In order to protect the usability of the system, however, we have a principle which we maintain: if a user schedules an action and the system merely pretends to accept it, users must be made aware of the deception prior to the time when the action is scheduled to occur. In practice, this is necessary, anyway, since certainly a user can discover that their action is not allowed to occur when they attempt it.

The difficulty with regard to usability is that it is not only the user who scheduled the initial action who will have inaccurate information in our scheme. Because actions depend on one another, it is quite possible that another low level user could schedule another later action which depends on the action which is not actually going to happen. In our previous example, Alice may have a coworker named Bob who is at the same security level. Perhaps Bob has made arrangements with Alice to use the data which is output by her supercomputer simulations.

Clearly, if Bob attempts to schedule an action which depends on Alice's action happening, then the system must not reveal to Bob that Alice's action will not be happening. As such, Bob's action must also be added to the cover story. When it comes out that Alice's action will not be happening if Bob still wants his to, then he may have to make additional arrangements. We wish to ensure that Bob has at least some chance to do this, which is why we have the principle that it must come out prior to the scheduled time for Alice's action.

This is feasible in certain cases because we are dealing only with classes of actions which are considered to be public actions, that is, although the scheduling of the action is sensitive, the action itself, by the time it is happening, is not.

7.3.3 Limitations

This approach is only feasible when a low secrecy action has its preconditions interfered with by a higher secrecy action. There is a second type of possible interference, which is when a low secrecy action interferes with the preconditions of a higher secrecy action which is already on the schedule at a later time. For example, let us say that Alice is working in one division of a company, and the CEO of the company, Carol, decided to assign Alice to a new division which the company was opening up on Wednesday. This new division will be working on a new product, so the assignment is a secret until its announcement. After Carol's action has been added to the schedule, Alice's immediate supervisor, Bob, does not feel that Alice is a good fit for his division and schedules for her to be fired on Tuesday. Obviously, if she were to be fired on Tuesday, she could not be reassigned to a new division on Wednesday, so Bob's action interferes with the preconditions of Carol's action.

This case cannot be dealt with via a cover story because Bob would have to know on Tuesday that his action would not be allowed to occur, thus revealing information about future plans. As such, this approach is useful in systems where lower secrecy users cannot schedule actions which invalidate the preconditions of actions taken by higher level users or where this sort of situation is dealt with through other means. We will discuss some other means which can be used to deal with this situation in Section 7.4.

But let us examine the restriction that lower secrecy users cannot schedule actions which invalidate preconditions of higher secrecy users' actions. Note that this condition restricts the effects and preconditions of actions based on which users are allowed to take them rather than the secrecy level of the action. This is convenient from a policy perspective since it does not require restrictions on which classes of actions are allowed to be high secrecy. But more than that, it is necessary because if a high secrecy user schedules a high secrecy action and then attempts to schedule a low secrecy action whose preconditions depend on the high secrecy action, this action must become high secrecy.

For instance, if the CEO schedules a restatement of a company's earnings, he might also schedule a meeting afterwards to talk to the employees about it. This meeting

would not be secret when it occurs or shortly beforehand, but obviously it should remain secret until the restatement of earnings has been made public.

Even though the action, unto itself, is not high secrecy, its existence implies the existence of a secret action, and so it must be treated as sensitive. As such, it becomes necessary that any action which can be taken by a high level user and which can possibly depend on a high secrecy action also be considered as high secrecy. This is why we describe the dependencies in terms of the level of the user attempting the action, rather than the level of the action itself.

This requirement about the relative actions and their preconditions and effects is relatively strong, but it only requires a partial separation of high secrecy and low secrecy users. As we have stated, a possible solution to the problems this paper addresses is to simply separate users into different security classes and ensure that there can be no dependencies at all across security classes. If this were carried out it would mean both that no actions which could be taken by high secrecy users could invalidate any preconditions for actions which could be taken by low secrecy users and that no actions which could be taken by low secrecy users could invalidate any preconditions for actions which could be taken by high secrecy users. Our condition is thus only a halfway separation of the classes, and, as such is much less stringent. This should be able to be attained in systems in which full separation of classes is not used.

This requirement is especially sensible in systems in which higher secrecy actions also tend to be higher priority actions. In section 7.4 we present another scheme which makes sense to use when higher secrecy actions tend to have lower priority than low secrecy actions. However, we do not have any approach which addresses the situation where the priority of high secrecy and low secrecy actions tends to be about the same.

Another limitation of this approach is that it cannot work effectively with an arbitary secrecy lattice, instead working best in a secrecy lattice which is a total order. This is because it is unclear what to do if your action conflicts with another action whose security label is incomparable to yours. If two actions with incomparable security labels conflict, the only thing which can be done which is consistent with the principles presented here is to remove whichever one is scheduled to occur later from the real schedule and then make it part of a cover story. Any other approach will either allow information to leak or will violate the property that users be allowed to know that their actions are part of a cover

story before they occur.

This behavior, however, is an arbitrary and capricious way of dealing with the conflict. It would function, from a security perspective, but if conflicts were frequent, large sets of apparently scheduled actions could simply fail to occur. Once the users saw the system state changing and figured out what had happened, this could result in users artificially pushing forward actions in order to assure that they really happen, but perhaps causing ill effects.

Instead, this approach is better suited to a situation in which there can be no conflict between actions with incomparable secrecy labels. On easy way to ensure this is to have the lattice be a total order, thus meaning that all actions are comparable. But it could also be achieved without a total order so long as the policies of the system were such that incomparable actions could not conflict. As long as there is an ordering between any two conflicting actions, one of the parties involved in any conflict can see that it's occurring and is able to make changes which might mitigate any problems. For simplicity, in the rest of this section, we will assume that the security lattice is a total order.

7.3.4 Functional Details

Now let us discuss how cover stories will work in particular. The system must necessarily maintain multiple schedules of events. Generally there will be a master list of which actions are actually scheduled. Every security class will be allowed to see a subset of this list. Beyond that, each security class below the top one will have some list of cover story actions, which is to say actions which have been requested and are consistent with the schedule which they can see, but inconsistent with the portion of the schedule which is hidden from them. Users are also allowed to see the cover stories of classes below theirs because it will help them maintain consistency with those schedules in their communications with those users and because it does not leak any information.

Speaking generally, there are two types of actions which can occur. First, a user can attempt to schedule an action whose secrecy is at the level of the user. In this case, we must consider whether the action is consistent with the real schedule and whether or not it is consistent with the schedule visible to the user. If the user is of the highest level, then these two questions are the same and things are very straight-forward. In that case, the action is added to the schedule if it is consistent with the real schedule and rejected

otherwise.

If, however, the user is not at the highest level, it could be the case that the schedule visible to the user is a cover story which contains actions which are not in the real schedule. In that case, we have four possibilities for the status of the action which the user is attempting to schedule:

- *Consistent with the cover story and the real schedule* In this case, we should simply add the action to the schedule.

- *Inconsistent with the cover story and the real schedule* In this case, we should simply reject the action.

- *Inconsistent with the cover story but consistent with the real schedule* Clearly we must tell the user that we have rejected his attempt because to do otherwise would reveal the existence of the cover story. In this case, if we reject the action then we are losing some potential functionality since the action is within the bounds of acceptability. However, accepting the action without telling the user could be significantly problematic. For example, upon failing to schedule an action the user would quite possibly attempt to schedule the same action for another time. If there were a series of silent successes, the results could be quite wasteful. As such, we argue that the appropriate thing to do is to simply reject the action.

- *Consistent with the cover story but inconsistent with the real schedule* In this case, the action must be added to the cover story. If the user's action were to appear to be rejected, this would compromise the cover story. As such, the action must be added to the cover story at the user's level.

The second possibility is that a user can attempt to schedule an action at a level below his own. If the action is inconsistent with the user's perceived schedule then it will not be allowed. If it is consistent with the user's perceived schedule, but not with a cover story at the level at which the user is trying to schedule it, then it will have its secrecy elevated. Specifically, its secrecy should be increased to the lowest level at which it is consistent with the perceived schedule for that level and all levels above that one and equal to or below the scheduling user.

Change Over Time

As time passes in the system, certain events will occur. Once they occur, it is assumed that they will become public knowledge. As this happens, the cover stories will need to change. Specifically, as secret actions are revealed, two things will need to occur. The first is that actions which will not occur (that is, actions which are only part of the cover story) should be removed. The second is that actions which depend on the secret actions may be revealed. Scheduled actions whose secrecy classes were artificially increased should have their secrecy class decreased as the higher secrecy actions on which they depended occur.

In order to accomplish these tasks, we would begin with the highest security class for which a cover story exists and work our way down, redoing the cover story for each security class in turn. In each class, we change the current state to reflect the action and then we begin moving through the scheduled actions from soonest to latest. We can now project a new state at any given time. If an action is not consistent with the new state, then we should remove it from the cover story. If an action was hidden due to its depending on a secret action which has occurred or conflicting with an action in the cover story which has been removed, then it should be revealed, so long as it has also been revealed in the secrecy classes above this one. If it is inconsistent with the one or more of the secrecy classes between its current secrecy level and the level we are presently changing the cover story for, then we should not add it to the current cover story because that would leak information to a higher level.

7.3.5 Proof of Indistinguishability

Let us be given a system, X, and an opponent, O, such that there is at least one scheduled action in X that O is not allowed to know about. That is, s is not empty. Let us consider the system identical to X except that s is empty, which we will call $X_\emptyset$.

Now let us assume that our necessary assumptions for the cover story scheme to work hold. Firstly, we assume that our security lattice, L, is totally ordered. Secondly, we assume that the security policy function P does not make decisions which depend on the presence of secret events. Note that if P factors secret events into its decision making, then no scheme can hide that information. Formally, we assume that for any schedule S,

$P(u, t, \sigma, S, A) = P(u, t, \sigma, S \bigcup s, A)$. Note that this does not require that the scheme allow actions which interfere with the secret scheduled actions, but rather just that it not rule them out using the policy function.

Thirdly, we assume that there is no possibility that a lower level user can schedule an action which would invalidate the precondition of a higher level scheduled action. Formally, this means that for all reachable states, (t, σ, S), there exists no user, u, and set of action, time pairs, A, such that $P(u, t, S, \sigma, A)$ is true and there exists some $s = (a_s, t_s) \in S$ such that $a_s.c(\sigma(S \bigcup A, t_s, \sigma, t))$ is false and $\hat{L}(u)$ does not dominate $\hat{L}(s)$. Note that this is not the same as saying that every action allowed under P is consistent with the existing schedule.

By definition, $X_\emptyset$ has a set of scheduled actions $S_\emptyset$ and every item in this set has a label which is dominated by the label of some user in O. X has a larger set of scheduled actions S, but the subset of X's scheduled actions which have a label dominated by a label of a user controlled by O is also $S_\emptyset$.

Given any request to schedule an action, P is the same for both X and $X_\emptyset$ and σ_0 and t_0 are identical, so, utilizing our second assumption, actions requested prior to the point of differentiation will be accepted or rejected uniformly by both systems.

Knowing that the policy function returns the same result only gets us halfway there. The other half is how the system handles action scheduling requests based on consistency. Actions inconsistent with both $S_\emptyset$ and S will rejected by both systems. Actions consistent with both $S_\emptyset$ and S will be accepted by both systems. Actions inconsistent with $S_\emptyset$ but consistent with S will be rejected by both systems. Actions consistent with $S_\emptyset$ but inconsistent with S will appear to be accepted by both systems. In truth, $X_\emptyset$ will accept the action, but X will only appear to accept it, adding it to its cover story.

As such, as events are added, $S_\emptyset$ will be identical to the cover story of X and as new events are added, the above will continue to be true. As such, we can know that the communication experienced by O for X and $X_\emptyset$ are identical.

Given two systems X_1 and X_2 which meet the preconditions of the indistinguishability definition, they will have a common $X_\emptyset$. Each of them will have identical communication to $X_\emptyset$ and hence also to each other. Thus, utilizing theorem 1, we know that systems using this scheme are completely indistinguishable when the assumptions outlined above hold.

7.4 Rescheduling

In the previous section we have detailed a complex scheme for ensuring the appearance of consistency. However, there is a much simpler scheme which can also prevent the detection of higher priority actions. Specifically, when a conflict occurs, we resolve the conflict by rescheduling or canceling the high secrecy scheduled action. This is to say, if a low secrecy user attempts to schedule an action which appears to that user as though it should succeed based on their knowledge of the schedule, then the attempt should always succeed. In order for it to succeed, we simply reschedule or cancel any scheduled actions of higher secrecy which interfere with it.

This approach can be used to prevent all inferences and works in any situation when the lattice is a total order. Also, like the cover story, it can work when the lattice is not a total order, so long as conflicts between incomparable actions will not occur. However, as is likely obvious to the reader, it carries significant drawbacks. The most immediate drawback is that it can lead to a great deal of inconvenience for high security users in terms of getting secret actions to occur when they wish them to. If a high secrecy action is scheduled, it may be preempted by a lower secrecy action at any time. As such, this scheme is really only sensible where higher secrecy actions are also lower priority actions. For example, if a company has a project which it is trying to keep secret, but which does not require completion in a timely manner then it might be sensible to schedule secret actions associated with the project, but to allow them to be preempted in order to maintain the secrecy.

The slightly more subtle problem is that it opens up a denial-of-service attack on higher secrecy users. A sufficiently knowledgable low level user could potentially schedule actions such that a particular high secrecy action could never occur. This attack is possible so long as there exists a low secrecy action which has preconditions which are invalidated by the high secrecy action which the user is trying to prevent.

The manner in which this scheme works is that when a user attempts to schedule an action, if that action is consistent with all of the scheduled actions visible to them, then the system must accept the action. If higher secrecy actions which are inconsistent with the action which is being scheduled exist than those must either be removed or rescheduled. There is guaranteed to be some solution, since the complete removal of all scheduled actions

with a higher secrecy than the newly scheduled action will always result in a consistent system.

In many case, higher secrecy actions could simply be rescheduled rather than removed, which is a much less drastic solution. However, finding an optimal rescheduling or removal of actions (measured by least perturbation of when actions occur or least number of scheduled actions removed) is an NP-Hard problem.

Proof 4 *This can be shown via a reduction from the k-Independent Set problem. Given a graph* (V, E)*, we can construct a schedule in which there is a scheduled action for each vertex in* $V = \{v_1, v_2, \ldots, v_n\}$*. Let our system have start time,* $t_0 = 0$*. Our system state will be composed of a set of boolean variables, one for each member of* V*,* $\{b_1, b_2, \ldots, b_n\}$*. The initial state,* σ_0 *will be the state such that all boolean variables are set to true. There are three types of actions. There are* n *of the first type which we shall designate* a_i^f *which have no precondition and, as an effect, set boolean variable* b_i *to false. There are also* n *of the second type which we shall designate* a_i^t *which also have no precondition and, as an effect, set boolean variable* b_i *to true. The third type consists of a single action* a^* *which has no effects, but has, as a precondition a boolean expression of the form* $\neg(\bigvee_{\forall i,j|(v_i,v_j)\in E}(b_i \wedge b_j))$*.*

The set of scheduled events $S = \{(a_i^t, n+i)\ \forall\ 1{\leq}i{\leq}n\}$*. All scheduled events are secret. We let our security lattice* $L = \{private, public\}$ *where private dominates public. For all* $s \in S$*,* $\hat{L}(s) = private$*. There is one user,* u*, and* $\hat{L}(u) = public$*. Our policy* P *is a function which evaluates to true for any arguments.*

A user u *such that* $\hat{L}(u) = public$ *submits a request to schedule the following set of actions:* $\{(a_i^f, i) \forall 1 \leq i \leq n\} \bigcup \{(a^*, 2n + n^2 + 1)\}$*. The minimum rescheduling for this schedule will tell us the minimal independent set.*

In order to add the requested set of actions to the schedule, it must be the case, that by the time the a^* *action is scheduled to occur, the system state does not have two boolean variables which are both true which correspond to vertices in the graph which share an edge. Thus, the condition of* a^* *is that all boolean variables which are still true share no edges between them and, as such, correspond directly to an independent set.*

We can either frame the problem in terms of the set of scheduled actions to remove or to reschedule. In this construction, the answer will be the same, since removing an action and moving it by slightly more than n *time units will have the same effect on the state at the point when the* a^* *action occurs. Although, it is also possible to move a scheduled*

action until after a^* this would cost at least n^2+1 time units, and hence never be part of an optimal solution. In either case, the optimal solution will be found by moving or removing the minimum number of nodes, thus leaving the nodes corresponding to a maximum independent set in place.

7.4.1 Proof of Indistinguishability

Fortunately, the security of the system does not depend on optimality of the rescheduling.

Let us be given a system, X, and an opponent, O, such that there is at least one scheduled action in X that O is not allowed to know about. That is, s is not empty. Let us consider the system identical to X except that s is empty, which we will call $X_\emptyset$.

Now let us assume that our necessary assumptions for the rescheduling scheme to work hold. Firstly, we assume that our security lattice, L, is totally ordered. Secondly, we assume that the security policy function P does not make decisions which depend on the presence of secret events. If this is the case, then no scheme can hide that information. Formally, we assume that for any schedule S and any set of secret events s' which can be derived from that schedule for X, $P(u,t,\sigma,S,A) = P(u,t,\sigma,S\bigcup s',A)$. This is a little different from the assumption in the previous proof because in this scheme the contents of s may be modified. We wish to capture the idea that regardless of modifications the policy function will be independent of s. Note that this does not require that the scheme allow actions which interfere with the secret scheduled actions, but rather just that it not rule them out using the policy function.

By definition, $X_\emptyset$ has a set of scheduled actions $S_\emptyset$ and every item in this set has label a which is dominated by the label of some user in O. X has a larger set of scheduled actions S, but the subset of X's scheduled actions which have a label dominated by a label of a user controlled by O is also $S_\emptyset$.

Given any request to schedule an action, P is the same for both X and $X_\emptyset$ and σ_0 and t_0 are identical, so, utilizing our second assumption, actions requested prior to the point of differentiation will be accepted or rejected uniformly by both systems.

Knowing that the policy function returns the same result only gets us halfway there. The other half is how the system handles action scheduling requests based on consistency. Actions inconsistent with $S_\emptyset$ will rejected by both systems. Actions consistent

with both $S_\emptyset$ and S will be accepted by both systems. Actions inconsistent with $S_\emptyset$ but consistent with S will be rejected by both systems. Actions consistent with $S_\emptyset$ but inconsistent with S will be accepted by both systems. In $X_\emptyset$ the action will be accepted in a very straightforward manner. But, in X, the action will be accepted, but changes will have to be made in s in order to accommodate it. Nonetheless, it will be added to S and hence the relationship between $S_\emptyset$ and S will be preserved and new actions will also be accepted or rejected as described above. As such, we can know that the communication experienced by O for X and $X_\emptyset$ are identical.

Given two systems X_1 and X_2 which meet the preconditions of the indistinguishability definition, they will have a common $X_\emptyset$. Each of them will have identical communication to $X_\emptyset$ and hence also to each other. Thus, utilizing theorem 1, we can know that systems using this scheme are completely indistinguishable when the assumptions outlined above hold.

7.4.2 Combining Schemes

This scheme can also be used in combination with our other technique, cover stories. As you can see, the assumptions used in the two different proofs of indistinguishability are compatible. This introduces then the possibility that the two approaches could be used together.

One possibility would be to use cover stories when possible, but in situations where cover stories could not be used, to fall back to rescheduling. Alternately, scheduled actions could be tagged with something indicating their priority, thus enabling the use of cover stories when the lower secrecy events were also lower priority and the use of rescheduling when the lower secrecy events have the higher priority.

7.5 Related Work

The arrangement and management of future actions or events are increasingly needed in both commercial and military collaborative applications, e.g., resource reservation, meeting scheduling, collaborative calendars and dynamic battle plan coordination [52]. Much work has been done on producing a feasible and desirable schedule that accommodates all the planned actions or events [17]. Recently, researchers have started to investigate the

privacy and security implications when conducting collaborative scheduling. The majority of the work studies this problem from the perspective of distributed constraint optimization, i.e., how to generate a schedule without revealing the resource constraints of each individual user. The problem we consider in this chapter is in a setting that is more dynamic. In particular, instead of collecting a set of scheduling request from multiple users and then generating a schedule, our system accepts ad hoc scheduling requests and tries to schedule them right away. Thus, the possible information leakage comes from the system's response to a user's scheduling request and the dependency between different actions or events, as defined by security policies.

The security objective we propose in this chapter is closely related to the concept of noninterference, which specifies that sensitive data should not have effect on nonsensitive data [16, 32]. Information flow policies and noninterference have been extensively studied in the literature. Many refined definitions of noninterference security principles have been proposed for different types of programming models [31, 33]. Sophisticated programming languages have also been designed and implemented to enforce noninterference [40, 47, 35]. The focus of most work in information flow is to ensure that a program can never violate noninterference during its execution. A major technique is to label data objects with security classes and then perform static analysis to show the noninterference property of a program. Besides considering information flow in a more specific system setting, this chapter takes a dynamic online approach to enforcing noninterference. Namely, we do not assume a program that defines the possible evolutions of a system. Instead, in our system, users may take actions at will and these actions will interfere with each other. The goal of this work is to let the reference monitor dynamically adjust the state of the system so that such interference cannot be observed by low-level users.

The idea of managing cover stories in multilevel secure databases has been examined in previous research [50, 23, 24]. When there exist tuples in a database table which have different security labels, certain problems can occur. If a low-level user attempts to overwrite the high-level data, this attempt cannot simply be denied because denying it would serve as a covert channel. However, allowing them to actually overwrite it could lead to denial-of-service attacks. Instead existing research shows that this problem can be addressed by polyinstantiation, that is, creating different versions of tuples for users of different levels. In some ways this is quite similar to our approach, but in databases the

different tuples are assumed to be independent of each other. In our scheduling system, different actions can be highly interrelated and this interrelation is the main potential source of information leakage. As a result, the research relating to databases only provides us with a jumping-off point.

Chapter 8

Conclusion

Organizations naturally tend to impose obligations upon the people working for them. As the actions which people take increasingly involve computers, it becomes sensible to let the computers help with the management of these obligations. However, this brings about new challenges in terms of balancing usability with security and ensuring that systems will function properly.

In this thesis, we have outlined a system for assigning and monitoring user obligations. This system and its components and their function have been described at length. In describing and investigating such a system we have discovered and are addressing a number of research problems.

The first problem addressed is how we ensure that a system will be likely to function properly while still enforcing the security policy of the system. In order to describe what is needed, we defined the property of accountability, which is when a system is in a state such that all of its obligations will be carried out so long as all of its users attempt to carry out their obligations. Although accountability sounds simple on the surface, we explored the difficulty in determining whether or not an arbitrary system is in an accountable state.

We explored the accountability problem and came to several conclusions. First we further refined the problem by defining what weak and strong accountability mean. Then we showed that in the most general case, both strong and weak accountability are undecidable. Next we introduced a set of assumptions about the system which allowed us to solve both the strong and weak accountability problems and showed that a relaxed set of assumptions would not be sufficient to guarantee that it would be solvable.

Next we introduced a simple concrete system which is sufficient to model many systems, and we demonstrated that solving the strong accountability problem in such a system is, in fact, tractable. Unfortunatly, the weak accountability problem was Co-NP hard, and we demonstrated that.

However, to take a moment, we would like to note that the negative results should not be considered too discouraging. Solving the accountability problem exactly is indeed a difficult task, however, in practice it should be possible in many systems to ensure that a system remains in an accountable state without solving the accountability problem. The accountability problem is fundamentally about determining exactly which states are accountable and which are unaccountable. However, in many systems, there will exist sets of states which we can know to be accountable with only polynomial time tests. By staying within that set of states, we guarantee accountability even though we would potentially compromise some usability.

An analogy for this would be that if we had a program which was generating logic formulas, and we wished to generate only formulas which are satisfiable, this would not require solving the satisfiability problem. Instead, we would simply place constraints on our formulas (a simple, but restrictive, one being "no negative terms") such that all the formulas we would generate would be satisfiable. In our case, if we want our system to remain accountable, it is possible to restrict the set of states to those which we can determine to be accountable within polynomial time or using other similar criteria. The impact that this would have on system usability would hopefully be small, but it would depend on the particulars of the system involved.

A second problem we described was how to assign blame and responsibility when things go wrong. In an accountable system, it should be that case that we always know who to blame for a singly obligation failure. But because users are imperfect, even a system which strives to maintain accountability may not always succeed. Sometimes multiple failures may occur. As such, it is helpful to have systems which can determine where the blame lies when multiple failures occur.

However, when we investigated blame in user obligation management systems we quickly found that blame is quite complicated. In fact, it is easy to see that in situations where the interactions between obligations are complex, determining blame is not something which can be done a posteriori. As such, we instead have proposed a system in which we

assign responsibility in a dynamic and policy-driven, but a priori manner. This has an additional benefit to users, which is that users are able to know what the effects will be if they fail to fulfill their obligations.

Following that, we tackled issues relating to what happens in an obligations system when user actions are denied. Specifically we outlined a means by which a new plan which achieves similar results can be automatically generated. We verified that this planning system is practical in at least some security systems by using it to create plans for automatically generated test cases in a system which uses miniARBAC.

Further, we described and tackled the problem of information leakage in a planning system. Unfortunately, we have only found limited solutions to this problem, but they form a good set of first steps in a very difficult problem.

8.1 Future Work

As can be seen in section 2.4 where we describe the model of the user obligation management system, there are still several components to the system about which there is research to be done.

Most noteably, the failed obligation management subsystem and the obligation adjustment subsystem. These two subsystems, although different in purpose both perform similar tasks, and, as such, will likely be researched together. Currently, we are attempting to understand whether or not it is feasible to apply some of the same techniques which we used in our failure feedback system to obligation adjustment and failed obligation management. This research is still in its initial stages.

We have also been investigating the use of model checking to determine accountability. Although model checking is intractable in the worst case, in practice, it tends to run successfully in many real world situations. As such, it may provide a feasible form of accountability checking for systems for which specialized algorithms have not yet been developed.

We are also planning to explore alternate operational policies. It can be argued that any reasonable operational policy would have to, at minimum, provide accountability, but there are certainly stronger operational policies which could exist. A stronger policy might hold more guarantees that users will still be able to fulfill their obligations even when

not all other users fulfill theirs. As such, it could be beneficial to formalize other operational policies and to examine the problem of checking systems for compliance with them.

We expect to continue to explore these and other areas in the future as we continue the development of our user obligation management system.

Bibliography

[1] Ross J. Anderson. A security policy model for clinical information systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 30–43, Washington, DC, 1996. IEEE Computer Society Press.

[2] Elisa Bertino, Francesco Buccafurri, Elena Ferrari, and Pasquale Rullo. A logical framework for reasoning on data access control policies. In *Proc. 12th IEEE Computer Security Foundations Workshop*, pages 175–189, Mordano, Italy, 1999. IEEE Computer Society Press.

[3] Elisa Bertino, Silvana Castano, and Elena Ferrari. On specifying security policies for web documents with an XML-based language. In *Proc. 6th ACM Symposium on Access Control Models and Technologies*, Chantilly, VA, May 2001. ACM Press.

[4] Denis Besnard and Budi Arief. Computer security impaired by legitimate users. *Computers & Security*, 23:253–264, 2004.

[5] Claudio Bettini, Sushil Jajodia, Xiaoyang Sean Wang, and Duminda Wijesekera. Provisions and obligations in policy management and security applications. In *VLDB*, Hong Kong, China, August 2002. IEEE Computer Society Press.

[6] Claudio Bettini, Sushil Jajodia, Xiaoyang Sean Wang, and Duminda Wijesekera. Obligation monitoring in policy management. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society Press.

[7] Claudio Bettini, Sushil Jajodia, Xiaoyang Sean Wang, and Duminda Wijesekera. Provisions and obligations in policy rule management. *J. Network Syst. Manage.*, 11(3):351–372, 2003.

[8] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, London, UK, February 1998. Springer-Verlag.

[9] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[10] C. Bussler and S. Jablonski. Policy resolution for workflow management systems. In *Proc. Hawaii International Conference on System Science*, Maui, Hawaii, January 1995. IEEE Computer Society Press.

[11] H. Chockler and J.Y. Halpern. Responsibility and Blame: A Structural-Model Approach. *Journal of Artifical Intelligence Research*, 22:93–115, 2004.

[12] D. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *2nd International Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, January 2001. Springer-Verlag.

[13] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[14] Babak Sadighi Firozabadi, Marek Sergot, Anna Squicciarini, and Elisa Bertino. A framework for contractual resource sharing in coalitions. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004*, Yorktown Heights, New York, June 2004. IEEE Computer Society Press.

[15] Pedro Gama and Paulo Ferreira. Obligation policies: An enforcement platform. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, Stockholm, Sweden, June 2005. IEEE Computer Society.

[16] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, Oakland, CA, April 1982.

[17] Rachel Greenstadt and Michael D. Smith. Collaborative scheduling: Threats and

promises. In *Workshop on the Economics of Information Security*, Cambridge, UK, June 2006.

[18] P.P. Griffiths and B.W. Wade. An authorization mechanism for a relational database systems. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.

[19] D. Grossi, F. Dignum, L.M.M. Royakkers, and J.C. Meyer. Collective Obligations and Agents: Who Gets the Blame? In *International Workshop on Deontic Logic in Computer Science*, Madeira, Portugal, May 2004.

[20] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[21] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Washington, DC, 1997. IEEE Computer Society Press.

[22] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 474–485, New York, NY, USA, 1997. ACM Press.

[23] Sushil Jajodia and Ravi S. Sandhu. Toward a multilevel secure relational data model. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security: An Integrated Collection of Essays*. 1994.

[24] Sushil Jajodia, Ravi S. Sandhu, and Barbara T. Blaustein. Solutions to the polyinstantiation problem. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security: An Integrated Collection of Essays*. 1994.

[25] Lalana Kagal, Timothy W. Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society.

[26] Hiroaki Kamoda, Masaki Yamaoka, Shigeyuki Matsuda, Krysia Broda, and Morris Sloman. Policy conflict analysis using free variable tableaux for access control in web

services environments. In *Policy Management for the Web Workshop*, Chiba, Japan, May 2005. Policy Management for the Web Workshop.

[27] Apu Kapadia and Geetanjali Sampemane. Know why your access was denied: Regulating feedback for usable security. In *In CCS 04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 52–61. ACM Press, 2004.

[28] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[29] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. ACM Conference on Computer and Communication Security*, New York, NY, USA, November 2000. ACM Press.

[30] Ninghui Li, William H. Winsborough, and John C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 123–139. IEEE Computer Society Press, May 2003.

[31] H. Mantel. Possibilistic definition of security – an assembly kit. In *IEEE Computer Security Foundations Workshop*, July 2000.

[32] John McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. 1994.

[33] John McLean. A general theory of composition for a class "possibilistic" security properties. *IEEE Transactions on Software Engineering*, 22(1), January 1996.

[34] P. McNamara. Deontic Logic, 2006. Stanford Encyclopedia of Phylosophy. Available at http://plato.stanford.edu/entries/logic-deontic/.

[35] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *ACM Symposium on principles of Programming Languages (POPL)*, San Antonio, TX, January 1999.

[36] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *IJCAI*, pages 459–466, 2001.

[37] J. Scott Penberthy. Ucpop: A sound, complete, partial order planner for adl. pages 103–114. Morgan Kaufmann, 1992.

[38] C. Riberiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An Access Control Language for Security Policies and Complex Constraints. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2001.

[39] Tatyana Ryutov and Clifford Neuman. Representation and evaluation of security policies for distributed system services. In *Proc. DARPA Information Survivability Conference and Exposition*, Los Alamitos, CA, USA, January 2000. IEEE Computer Society Press.

[40] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[41] Martin Sailer and Michel Morciniec. Monitoring and execution for contract compliance. Technical Report TR 2001-261, HP Labs, 2001.

[42] R. Sandhu and P. Samarati. Authentication, Access Control, and Audit. *ACM Computing Survey*, 28(1), March 1996.

[43] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, February 1999.

[44] Ravi S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating systems. *Journal of the ACM*, 35(2):404–432, 1988.

[45] Ravi S. Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, February 1999.

[46] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 124–138, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[48] E.G. Sirer and K. Wang. An access control language for web services. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*, New York, NY, USA, June 2002. ACM Press.

[49] XACML TC. Oasis extensible access control markup language (xacml). *http://www.oasis-open.org/committees/xacml/*.

[50] D. Garvey Thomas and Teresa F. Lunt. Cover stories for database security. In *IFIP WG 11.3 Workshop on Database Security*, Shepherdstown, WV, November 1991.

[51] Andrzej Uszok, Jeffrey M. Bradshaw, Renia Jeffers, Niranjan Suri, Patrick J. Hayes, Maggie R. Breedy, Larry Bunch, Matt Johnson, Shriniwas Kulkarni, and James Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society Press.

[52] Tom Wagner. Coordination decision support assitants (coordinators). `http://www.darpa.mil/ipto/programs/coor/coor_concept.asp`.

[53] A. Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Monterery, CA, August 1999.

[54] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/, 2005.

[55] M.E. Zurko, R. Simon, and T. Sanfilippo. A User-Centered, Modular Authorization Service Built on An RBAC Foundation. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.