PRACTICAL USER OBLIGATION SYSTEMS THAT AFFECT AND DEPEND ON AUTHORIZATIONS

APPROVED BY SUPERVISING COMMITTEE:

Jianwei Niu, Ph.D., Chair

Ravi Sandhu, Ph.D.

Rajendra Boppana, Ph.D.

Jeffery von Ronne, Ph.D.

Ting Yu, Ph.D.

Accepted:

Dean, Graduate School

DEDICATION

"When you make the finding yourself - even if you're the last person on Earth to see the light - you'll never forget it." Carl Sagan

This dissertation is dedicated...

in memoriam of my mother Anna, *who will always be with me.*

in memoriam of my late supervisor Will, *who was not only a mentor, but also a very dear friend.*

to my wife Emanuelle, *the love of my life.*

to my sister Anna, *my best friend.*

to my grandma Conchita, *who always inspired me, even in my hardest times.*

to my father Clementino, *for your support and incitement.*

PRACTICAL USER OBLIGATION SYSTEMS THAT AFFECT AND DEPEND ON AUTHORIZATIONS

by

MURILLO DE BARROS COSTA REGO AMAZONAS PONTUAL, M.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2011

ACKNOWLEDGEMENTS

First, I would like to take the opportunity to acknowledge and extend my heartfelt gratitude to my late advisor, Dr. William H. Winsborough. Without his expertise and guidance, this research could not be accomplished. He has guided me and also challenged me during this journey. He has helped me learn not only how to approach a technical problem but also taught me how to be a better person in life. He always emphasized that contributing to science is more important than just publishing papers, which is one of my most important takeaways from Ph.D.

Second, I would like to thank my colleague, collaborator, and friend Omar Haider Chowdhury. His insights and expertise have helped me workout several technical details of this dissertation. Furthermore, he has also helped me proofread this dissertation.

Third, I would also like to express gratitude to my current advisor Dr. Jianwei Niu for believing in me. She has supported me when I needed it most.

Fourth, I would like to thank my collaborators Dr. Ting Yu, and Dr. Keith Irwin. They have been very helpful throughout this journey and have helped me articulate, formalize, and solve different problems related to my research.

Finally, I would like to thank my friends Andreas Gampe and Jeff McAddams who have helped me proofread this dissertation. In addition, I would also like to thank my committee members for spending their time on me.

December 2011

PRACTICAL USER OBLIGATION SYSTEMS THAT AFFECT AND DEPEND ON AUTHORIZATIONS

Murillo de Barros C. R. A. Pontual, Ph.D
The University of Texas at San Antonio, 2011

Supervising Professors:
Jianwei Niu, Ph.D.
William H. Winsborough, Ph.D.

Many authorization systems include some notion of obligations. However, most of the prior works concentrate on policy specification and enforcement of system obligations. Little attention has been given to user obligations that can depend on and affect authorizations. A user obligation is an action that a user must perform in some stipulated time window. As automated tools seek to provide increasing support for managing personnel and projects, there is an increasing need for individual tasks to be assigned and coordinated with authorization, and for supporting automated techniques. Thus, the management of user obligations that depend on and affect authorizations is a significant issue in the field of computer security. In this context, a user may incur an obligation that she is unauthorized to perform.

Prior work has introduced property of the authorization system state that ensures users will be authorized to fulfill their obligations in the appropriate times. We call this property *accountability* because users that fail to perform authorized obligations are accountable for their non-performance. Roughly state, a system state is accountable when each of the pending obligations in the current state is authorized no matter when all the other obligations would be performed in their associated time interval.

Thus, accountability property can be viewed as an invariant that the system attempts to maintain. This invariant of the system ensures that if the users are diligent, then they will be authorized to fulfill their obligations. To this end, it may be necessary to prevent discretionary (non-obligatory) actions being performed if they would violate the accountability property. We achieve this by augmenting the reference monitor to deny actions that violate it. The prior work is inconclusive and purely theoretical in regards to the feasibility of maintaining accountability in practice. In addition, there are several technical challenges and issues in designing such systems which the prior work overlooked.

In this dissertation, we develop a collection of techniques and tools to address these issues. First, we study the scalability of our abstract obligation model. To this end, we present an instantiation of the abstract obligation model by using simplified versions of the Role-based Access Control (RBAC) and the Administrative Role-based Access Control (ARBAC) (*i.e.*, mini-RBAC and mini-ARBAC) as its authorization system. Based on our empirical evaluations, we believe that our obligation model is efficient enough to be used in practice. Furthermore, in order to show the flexibility of our abstract model, we instantiate it with a simplified version of the HRU Access Control Matrix Model. We also compare the performance of these two instantiations through experiments.

Secondly, we enhance the usability of our obligation system by providing techniques that assist users to overcome authorization denials due to accountability violation. For this, we develop an approach based on an AI-planning tool that provides the user with an alternative plan of actions to achieve her goal. Our empirical results indicate that our tool can handle moderate sized problem instances.

Thirdly, we present techniques that provide administrators ways to overcome accountability violation. This is particularly useful when obligations are violated, users are reassigned to different divisions, or new projects or business functions are added.

Finally, we enrich our obligation model to support different kinds of obligations that occur in practice (*viz.*, repetitive and cascading obligations). When one obligation incurs another obligation, we call this phenomenon the "*cascading*" of obligations. Repetitive obligations are obligations that repeat after some predefined time. We provide techniques to decide the accountability property efficiently in their presence. Our experimental results show that accountability can be decided efficiently in presence of restricted versions of these kinds of obligations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Maintaining security in modern organizations depends on security procedures being faithfully carried out, both by computer systems and by humans. A computer system relies on authorization systems to prevent malicious or accidental violation of confidentiality and integrity requirements. Security policies can be used to guarantee that the authorization mechanisms are reflecting the desired security requisites. In general, a security policy is a formal or semi-formal specification that describes many different security aspects in an organization. Usually, security policy contains the configurations of the security mechanisms, how resources can be accessed, the permitted information flow, how data is stored, *etc.* In addition to these requirements, some security policies may also contain notion of obligations. An *obligation* defines an action that must be performed by some stakeholder in a future time. One of the most common type of obligations is the system obligation. A system obligation is an obligatory action that must be fulfilled by the system itself. For example, if a user fails more than three times to login into a system, then the system *must block* the user and *must send* an email to the administrator. However, not all the obligatory actions can be performed by automatic systems. In fact, security mechanisms also rely on human users and administrators performing actions that range broadly, including tasks such as business functions and administrative operations, and are also an obligatory part of the humans' job descriptions. We call these actions *user obligations*. For instance, a conference system could require a user (in the role of reviewer) to review a paper in no less than two weeks.

Most user obligations may also require corresponding system authorizations. This is due to the fact that a user obligation is nothing more than an action being performed in a computer system, and therefore being subjected to the same permission requirements (*e.g.*, if Alice has an obligation to check the log files; she needs to have the "read" permission to the log files) as any typical action. In addition, many obligations have an effect on the authorization mechanisms. For instance, when an employee is reassigned to a different division, an administrator may incur obligations to revoke the permissions of the transferred employee. As automated tools seek to provide increasing support for managing personnel and projects, there is an increasing need for individual tasks to be assigned and coordinated with authorizations, and for supporting automated techniques.

Nowadays, more and more government privacy regulations are being created, for instance, Children's Online Privacy Protection Act of 1998 (COPPA), Gramm-Leach-Bliley Act (GLB) of 1998, The Health Insurance Portability and Accountability Act (HIPAA) of 1996, etc. Many of them contain some notions of user obligations that depend on authorization. For example, the 45 CFR part 164 HIPAA [28] contains such user obligation, and states that an individual has the right to request a Covered Entity (CE) (e.g., a hospital) to amend his protected health information (PHI). After this request, the CE is obliged no later than 60 days to correct the individual's PHI (for doing this, some employee of the CE must have the appropriate permissions) or to provide the individual the reasons for not amending his PHI. Thus, the management of user obligations that depend on and affect authorizations is a significant issue in the field of computer security.

1.1 Terminology

In this section, we present different categories of obligations that exist in the literature. This is not intended to be an exhaustive presentation, instead it serves to introduce the reader to the obligation terminology. In general, an obligation contains the following components, an obligatory action, a principal that is obliged to perform the obligatory action, a time window when the obligatory action must be performed and finalized, and a condition that triggers the obligation. In this dissertation, we use the terms principal, stakeholder, and agent indistinguishably, to represent the agent that is obliged to perform an obligation. In this case, the obligatee can either refer to a human being or a machine agent. On the other hand, we use the terms user or subject to indicate that an obligation is being performed by a single human user.

Types of Obligations

Obligations are requirements that the obligatory agent needs to follow. Such requirements can require that the agent to take certain actions, or on the other hand, can forbid the agent to take certain actions. In addition to this, the obligatory requirement may need the agent to take certain actions before granting the access to a resource or other actions. Now, we present the three different forms of obligations:

1. **Obligations** (also known as positive obligations, post-obligations) are actions that a principal must perform in some stipulated future time. For example, Alice must check the log files in the first week of the next month.
2. **Refrainments** (also known as negative obligations, anti-obligations) are actions that a principal is forbidden to take in a particular time. For example, Bob is forbidden to check the log files in the next month.
3. **Provisions** (also known as pre-obligations) are actions that a principal takes in order to gain access to a resource. For example, in order to check the log files, Bob must request an authorization.

Types of Agents

Obligations can be carried out by humans or the system itself. Below, we present the types of obligations according to the type of the principals:

1. **System obligations** are fulfilled by the computer system itself. For example, the system must check the integrity of the log files in the first day of the next month.
2. **User obligations** are fulfilled by human user of the system. For example, Alice must check the log files in the first day of the next month.

Number of Agents

Depending on the number of agents who are obliged to fulfill an obligation, we have the following classification:

1. **Collective obligations** (also know as group obligations) must be fulfilled by a group of agents. Note that, a collective obligation may require that at least one agent performs the obligation, or it may require all the agents that are members of the group to perform the obligation. For instance, Alice or Bob must check the log files in the next month. Another example could be, Alice and Bob must check and digitally sign the log files in the next month.

2. **One agent obligations** are fulfilled by only one agent. For example, Alice must check the log files in the next month.

Types of Triggers

An obligation can be triggered by one of the following ways:

1. **Conditional Obligations** are triggered by some condition. For instance, as long as Alice is the security manager, she must check the log files every month.

2. **Event obligations** are triggered by events. The events can be controllable (internal), or can be a uncontrollable (external). For example, if the manager Alice requests Bob to check the log files(*i.e.*, a controllable event), then he must check the log files. Another example could be, if a server crashes (*i.e.*, a uncontrollable event), then Alice must restore it in no more than 10 hours.

Types of Deadlines

The deadlines of obligations can be defined in two different ways:

1. **Ongoing Obligations** are terminated when some event occurs. In other words, must be performed during a usage. For instance, Alice is obliged to check the log files until the project is over. Her obligation is terminated when the project is over.

2. **Time Interval Obligations** must be performed within a given time interval. For example, Alice must must check the log files between 7 AM and 9 PM.

3. **Immediate Deadline Obligation** must happen and finish just after the event that triggered it For instance, if there is an error on a transaction; the system must roll back all the actions.

Incremental and Non-incremental

Obligations can be fulfilled in an incremental or non-incremental way:

1. **Non-incremental Obligations** are seen as atomic unit. For instance, Alice is obliged to check the log files in the next week.

2. **Incremental Obligations** have an internal state and are fulfilled incrementally. For example, Alice is obliged to pay her bill in no more than three incremental payments.

Repetitive and Non-repetitive Obligations

Obligations can happen indefinitely in time, or can occur for a finite particular time. According to this terminology we can have the following classification of obligations:

1. **Non-repetitive Obligations** are obligations that happen only once. For instance, Alice is obligated to check the log files in the next week.

2. **Finite Repetitive Obligations** are obligations that recur at fixed time intervals in a finite way. For instance, Alice is obliged to check the log files in the first day of the month during one year.

3. **Infinite Obligations** that recur indefinitely at fixed time intervals. For example, every first day of the month Alice must check the log files.

Note that, Finite Repetitive Obligation is a particular case of the Ongoing obligation. However, both Finite and Infinite Repetitive Obligations can also occur in a non-periodic way.

1.2 Problem Statement

Work on computer managed obligations goes back several decades [34,39,40]. Many works focus on policy determination of obligations [2,5,12,15,23,38,42,55]. Relatively little work has focused on specification of the proper discharge of obligations [12,24,26,31,41]. Even less has focused on user obligations.

Working in the context of user obligations, Irwin *et al.* [31,32] were the first to study obligations that depend on and affect authorizations. A correctly functioning system has a regular behavior, whereas a human user's behavior cannot be predicted or enforced. This situation raises complexity in designing and analyzing a user obligation system. One such situation arises when the user Alice is assigned to perform an obligation in July next year, for which she currently does not have the proper authorization. A traditional reference monitor will not deny the addition of such an obligation. If some other user does not first give Alice the proper authorization, her request to perform the obligatory action will be denied. No one may realize that she can not fulfill her obligation until she attempts to perform it. Irwin *et al.* observe that deadlines are needed for user obligations in order to be able to capture the notion of violation of obligations. Associating a start time as well as an end time with each obligation, Irwin *et al.* introduce properties called *strong* and *weak accountability* [31] that ensure that each obligatory action will be authorized during its stipulated time interval. For a given authorization policy, accountability is a property of the authorization state and the obligation pool. Roughly stated, accountability holds if each obligatory action will be authorized, no matter when all the other obligations are fulfilled in their associated time intervals. The strong version of accountability requires that each obligation be authorized throughout its entire time interval. On the other hand, weak accountability allows an obligation to be unauthorized during part of its time interval. However, it ensures that if the obligated user waits for other obligations to be fulfilled, it is guaranteed that the action will become authorized in its deadline. Other types of accountability exist [30]; however they are out of the scope of this dissertation.

Accountability properties are important because they allow users to reason about future states of the system. In the scenario presented above, assume the administrator Bob tries to add an obligation to Alice that she needs to perform in July next year. If she does not possess the permissions to fulfill her obligation, and there is no obligation (in the set of pending obligations) that gives such permissions to her, then Bob will be notified by the system that he cannot add this obligation, as doing so would compromise accountability of the state. A big advantage of this type of system is that the administrators do not need to wait for users trying to execute their obligation, in order to discover that the users do not have the permissions. Thus, accountability properties can be viewed as invariants that the system attempts to maintain. To this end, it may be necessary to prevent discretionary (non-obligatory) actions being performed if they would disturb accountability.[1] To accomplish this, a user obligation system can have a reference monitor that is responsible for denying actions that violate the accountability. A discretionary action can violate the accountability property in one of the following ways:

1. The action is administrative and can make an obligated user unable to perform her obligation.

2. The action causes an obligation to be incurred that will not be authorized.

3. The action introduces an administrative obligation that will make an obligated user unable to perform a subsequent obligation.

The results presented by Irwin *et al.* [31,32] were inconclusive from the standpoint of practicality. They show that when the authorization system used in the obligation model is fully abstract, accountability is undecidable. This follows from the fact that the authorization system can perform arbitrary computations. When it is instantiated with a simplified HRU Access Control Matrix Model, as is done by Irwin *et al.* [31, 32] , the problem of determining strong accountability becomes polynomial. However, for the algorithm they present, this polynomial has degree four in the number of obligations and degree two in the policy size. Additionally, no performance evaluation is provided to determine the limit of problem instance size that can be efficiently handled. Moreover, determining weak accountability in the instantiated case is intractable (co-NP hard) and no decision procedure is presented.

[1]In many cases it will be necessary to enable users (especially administrators) to force discretionary actions to be performed, even when doing so violates accountability. Also, since it is impossible to ensure that users will fulfill their obligations, accountability guarantees obligatory actions will be authorized only under the assumption that other obligations on which they depend are faithfully discharged.

Irwin's abstract model supports the notion of cascading obligations, by which we mean that obligatory actions can cause additional obligations to be incurred. However, when they instantiate their concrete model with to use a simplified HRU Access Control Matrix Model, they disallow cascading obligations. This is because several issues make cascading obligations difficult to handle correctly. More specifically, three problems make supporting cascading difficult and unproductive with their model:

1. Different policy rules can cause different (disjunctive) obligations to be incurred, making it computationally expensive to reason about the future state of the obligation pool and authorization system.

2. Cycles can easily be formed that introduce the likelihood of infinite sequences of new obligations being incurred as the result of a single action.

3. The time intervals during which new obligations are to be performed depend on the time at which the action that causes them to be incurred is performed. As obligations are scheduled times further from the current time are considered, the time intervals in which obligatory actions could occur become longer. This makes it increasingly unlikely that these obligations will be authorized throughout the entire interval in which they must be to satisfy accountability.

Another problem that was not addressed in previous work [31, 32] is how obligation systems handle the violation of obligations. Recall that accountability properties guarantee that users will have the necessary permissions to perform their obligations; considering that all the users diligently fulfill their obligations. However, when a user is unable to fulfill his obligations, then all other obligations that depend on this violated obligations may also be violated. For instance, consider an obligation system that has two users, Alice and Bob. Bob is a project manager, and Alice is a member of Bob's team. Now, consider that Bob has an obligation to grant Alice the role of developer, and Alice has an obligation to develop source code. Let us assume that Bob was ill and was unable to fulfill his obligation. Due to Bob's illness, Alice did not receive the developer role that she needs to create source code. Thus, lack of authorization will cause Alice to violate her obligation. Because human behavior is unpredictable, it is clear that when some obligations are violated, a system must provide some strategies to assist the administrator to recover the system from these violations.

1.3 Objective and Thesis Statement

Based on the discussion above, it is clear that there are many research issues that must be addressed by any system that manages obligations that interact with authorization. This section summarizes the ones that we address in this dissertation.

Scalability and Performance A real life obligation system should be scalable enough to support a large number of obligations, users, and objects. In our case, we want to investigate whether it is possible to efficiently compute both versions of accountability in a way that these decision procedures can be added to a reference monitor. Recall that every time a user tries to execute an action; the system needs to check whether the action is authorized and it is not violating accountability (*i.e.*, removing the permission of other obligations). Thus, if a user needs to wait more than few seconds when executing an action; it is clear that such a system is not going to be very usable in practice. Previous work provided a higher cost polynomial algorithm for strongly accountability, but did not provide any decision for weak accountability. In this dissertation, we present efficient decision procedures for both versions of accountability.

Flexibility In this dissertation, we want to show that our abstract obligation model is flexible enough to support different authorization systems. For this, we want to instantiate it with two different authorization models, namely, mini-ARBAC/mini-RBAC and a simplified version of the HRU Access Control Matrix Model.

Usability Recall from the previous section that even when a user has the proper authorizations, his action may be denied if it violates accountability (*i.e.*, an action is revoking other obligations permissions). Previous work [21, 36] have presented techniques for providing users with assistance in understanding and overcoming authorization denials. However, only Irwin *et al.* have presented techniques to overcome these denials when considering user obligations and accountability. Because of privacy concerns, as well as the intricate interactions between actions and pending obligations, Irwin's technique presented the user a plan of actions that will allow him to execute his desirable action without violating accountability. Irwin's technique was based on AI planning, but no complexity analysis and only with partial empirical evaluations were presented. The aim of this dissertation is to provide a formal description of this problem, complexity analysis, and a comprehensive empirical evaluation of this approach.

Support for Different types of Obligations An obligation system must have an appropriate balance between efficiency and expressive power (managing different types of obligations). Irwin *et al.* presented a polynomial time algorithm for determining strong accountability when their obligation model was instantiated with an access control matrix model as its authorization state. To accomplish this, they made an additional assumption, which prevents cascading of obligations. As depicted in previous section several issues complicate handling such situations. In fact, Irwin *et al.* have proved that deciding accountability problem without any restriction when using their abstract model is undecidable. However, to be usable, an obligation system must be able to express many different forms of obligations that occur in the real world, as for example, cascading of obligations, finite and infinite repetitive obligations, *etc.* In this dissertation, we explore restricted notions of cascading of obligations, and we present efficient procedure for deciding accountability in presence of them.

Violation Recovery As mentioned in the previous section, the idea of preserving accountability as a system invariant sometimes might not be achieved, as users may fail to fulfill their obligations. Furthermore, it may be necessary to revoke a user's authorization, as when a user leaves an organization or is transfered to a different position. In such cases, tools are needed that enable administrators and other authorities to remove or reassign obligations, or to add new obligations that replace old ones in a manner that authorizes new or existing dependent obligations. These may include tools that analyze dependencies among existing obligations or that propose plans by the help of which accountability could be maintained or restored.

Thesis Statement The primary goal of this dissertation is to identify, specify, analyze, and solve several fundamental technical challenges that arise in designing a practical user obligation system that can affect and depend on authorization. In our scope, an obligation is a positive, non-incremental, user obligation, triggered by an event, fulfilled instantly and with deadlines. (Pre-obligations, negative obligations and conditional obligations are out of the scope of this thesis.)

1.3.1 Approach

The techniques we develop are the follows:

1. We instantiate the authorization portion of the obligation model of Irwin with a previously studied simplified version of the administrative role-based access control model [48, 50] called mini-ARBAC and mini-RBAC [51]. To this end we present:

   (a) A more efficient algorithm that determines strong accountability of a set of pending obligations under this instantiated model. This algorithm has the complexity $n^2 \log n$ times the policy size, in which n is the number of pending obligations. The empirical evaluation that we have conducted shows that the algorithm runs in less than 8 milliseconds, even for very large policies and obligation sets[2], and that there are cases in which each one outperforms the other. In short, the special-purpose algorithm can handle larger problem instances (more obligations, users, roles, and policy rules), when the algorithm is used incrementally to determine whether a single obligation can be added to an accountable obligation pool.

[2]In this dissertation, we use the terms small obligation pool size for describing obligation pools that have less than 200 obligations, moderate obligation pool size when we have more than 200 and less than 2000 obligations, and large obligation pool size when we have more than 2000 obligations. Furthermore, a small-size policy concerns less than 50 users and 50 objects, moderate-size policy concerns less than 200 users and 200 objects, and large-size policy concerns more than 200 users and 200 objects.

(b) We provide a proof that deciding for weak accountability is co-NP complete in the simplified model, and remains so when only one policy rule enables each action and the condition expressed by that rule is purely conjunctive. (Both positive and negative role memberships can be tested, however.)

   i. We explore two approaches for weak accountability, one that designs an algorithm specifically to solve this problem, and one that uses model checking. We provide, design, specification, optimization, prototypes, and empirical evaluations of these techniques. Our empirical evaluation of these techniques indicate that they are effective in many cases for obligation sets and policies of moderate size, provided interdependent obligations do not overlap too much, while the model checker is much better able to deal with the many possible interleaving of overlapping obligations.

(c) We also give a reformulation of Irwin's obligation system and accountability definitions that makes some improvements on the original. The most significant of these is that we formalize the scheduling of obligations in terms of traces (sequences of states and actions), rather than by assigning times at which actions are performed. Assigning times raises the issue of two actions being scheduled at the same time. Since actions must be atomic, Irwin *et al.* introduced a fixed, arbitrary order on actions that defines the order in which actions scheduled for the same time are performed. Not only does this make the presentation cumbersome. This order can actually affect whether a given obligation pool is accountable. In the current formulation, time is used only to define the relative order in which obligations may be carried out.

2. We improve the algorithm presented in [31] that uses a simplified HRU Access Control Matrix Model as authorization state to utilize the same data structure we developed in [45]. Then, we present an empirical evaluation comparison between the strong accountability algorithm under the mini-ARBC model and a simplified HRU Access Control Matrix Model.

In a further additional result, we show that the weak accountability decision problem remains co-NP complete when the authorization system is restricted to use only the expressive power of the original HRU Access Control Matrix Model.

3. Sometimes, preserving accountability as a system invariant may not be achieved. For example, users may fail to fulfill their obligations. Furthermore, it may be necessary to change a user's authorization to fulfill their obligations, as when a user leaves an organization or is transferred to a different position. In this dissertation, we provide techniques to be used by obligation system managers to restore accountability, such techniques may include removal or reassignment of obligations, or to add new obligations that replace old ones in a manner that authorizes fulfillment of new or existing dependent obligations. We introduce several notions of dependence among pending obligations that must be considered in this process. We also introduce a novel notion we call *obligation pool slicing*, owing to its similarity to program slicing. An obligation pool slice identifies a set of obligations that the administrator may need to consider when applying strategies proposed here for restoring accountability.

4. We also present a precise description of the action failure-feedback problem. In this approach, when a user attempts to execute an action that will be denied (either it or the obligation incurred violates accountability), we try to present the user an alternate plan of actions that will enable her to perform the desired action. We consider that this approach can be particularly useful in project management environments or in case of workflows, when it is desirable to ensure that a user is going to be able to fulfill her main task, even when she has some particular actions denied.

    (a) We study the question whether it is possible to have an action failure feedback component in practice. To answer this question we utilize an approach presented by Irwin in his PhD dissertation [30]. The approach uses an AI planning technique to solve this problem. The contributions of my dissertation are:

        i. We provide a precise formal definition of the action failure-feedback problem.

   ii. We also prove that the action failure-feedback problem is PSPACE-hard for a user obligation system that uses mini-RBAC and mini-ARBAC as the authorization model.
   iii. Given the difficulty of obtaining real life policies and obligations; we present three different strategies to generate complex problem instances in order to assess the limits of Irwin's approach in practice.
   iv. We also present empirical evaluations of Irwin's approach. It indicates that the approach is effective for small sized obligation sets and policies.

5. As described before, cascading obligation is an important topic on the studies of obligation. In this dissertation, we aim to support restricted forms of cascading obligations in our concrete model and to provide techniques and tools that can decide accountability in presence of them. them. To this end, we present the following:

   (a) A cascading obligation taxonomy.

   (b) We show that deciding strong accountability when considering unrestricted cascading obligations, using our concrete model using the mini-ARBAC/mini-RBAC as its authorization system is NP hard.

   (c) We present algorithms and empirical evaluations for two restricted versions of cascading obligations, namely, non-infinite cascading, and infinite repetitive obligations. The empirical evaluations of these algorithms indicate that they can decide strong accountability in less than $1/10^{th}$ of a second, even for very large obligation sets.

1.3.2 Contributions

The contributions of this dissertation are summarized below:

1. Algorithms, complexity analysis, and empirical evaluations for strong and weak accountability properties using mini-RBAC and mini-ARBAC.

2. Comparison between strong accountability technique using mini-RBAC and mini-ARBAC, and strong accountability technique using a simplified version of the HRU Access Control Matrix Model.

3. The complexity analysis of the action failure feedback problem, and its empirical evaluation.

4. A comprehensive study of techniques that can be used to restore accountability when accountability is violated.

5. Algorithms, complexity analysis, and empirical evaluations for strong accountability when considering restricted version of cascading obligations using mini-RBAC and mini-ARBAC.

1.4 Outline

The reminder of this dissertation is organized as follows. Chapter 2 provides our obligation architecture, abstract, concrete models as well authorization states. Chapter 3 presents the definitions of accountability, techniques and their empirical evaluations. Chapter 4 provides the action failure feedback problem, techniques and their empirical evaluations. Chapter 5 discusses the techniques to restore accountability. Chapter 6 presents the management of handling cascading obligations. Chapter 7 discusses the related work. Chapter 8 provides the conclusion and future work of this dissertation.

CHAPTER 2: USER OBLIGATION MODEL

In this chapter, we introduce our framework for managing user obligations that can affect and depend on authorization. The architecture of the obligation system is inspired by the model presented by Irwin *et al.* [30–32]. As stated before, our obligation system supports user obligations that are triggered by events, have explicit time windows, and are atomic in nature. After presenting the architecture of the obligation system, we then show the formal aspects of our abstract and concrete models. For the concrete model, we use mini-RBAC and mini-ARBAC as the authorization system of our choice. Roughly speaking, mini-RBAC (*resp.*, mini-ARBAC) is a simplified version of the widely used RBAC (*resp.*, ARBAC) system [49]. In RBAC, users are given authorizations based on the role they have. The main differences between mini-RBAC and mini-ARBAC with their standard counterpart are as follows; they do not support sessions, explicit separation of duties, role hierarchies, and disjunction of roles in the pre-condition. A more comprehensive discussion of mini-RBAC and mini-ARBAC is provided in section 2.5.1. In addition, we also present a concrete model instantiated with a simplified version of the HRU Access Control Matrix Model. Note that, we do not consider this instantiation as a novel contribution as it was previously presented in Irwin *et al.* [31]. However this is necessary to understand the contributions in the next chapter. The following chapters of this dissertation use the concrete models presented in this chapter.

2.1 Events

Before introducing our obligation framework, let us examine how obligations are incurred in our model. We assume that obligations are triggered by events. An event is an action that may trigger a state change in the system. We consider two classes of events in our system, namely, *controllable events* and *uncontrollable events*. Controllable events are originated by actions taken by users of the system on zero or more objects. (*E.g.*, when Alice creates a report, she may incur an obligation to later submit the report to her supervisor. Presumably creating the report is a controllable event.) Note that the action part of a controllable event can either be a discretionary action or an obligatory action. On the other hand, uncontrollable events are generated by the environment. (*E.g.*, a policy might require that when a court order arrives, a company lawyer must review and respond to it. Receiving a court order is a uncontrollable event.) In this dissertation, we only consider controllable events.

2.2 Obligation State

At each point in time, an obligation can be in one the following states: pending, fulfilled, or violated. We say an obligation is *fulfilled* if the action identified by the obligation has been executed during the obligation's time interval. We say an obligation is *violated* when the action defined by the obligation was not executed in its proper time interval. Finally, we say an obligation is *pending* if the current time is less than the end time period of the obligation and the obligation has not yet been executed. A pending obligation can be in one of the two states, available or unavailable. We say an obligation is *available* when the obligation is pending and all the resources, authorization and users required to fulfill it are available. On the other hand, an obligation is *unavailable* when it is pending and one of the following is not available during the its entire time interval: authorization, user, or resources. However, the principal interest of this dissertation is to guarantee authorization availability. (Ensuring user availability and resource availability are matters for future work.)

2.3 General Architecture

This section presents our architecture [46] for managing user obligations that can depend on and affect authorization (figure 2.1). In figure 2.1, the arrows represent messages that can be exchanged among software components and users. The direction of the arrow indicates the direction of message flow. Let us now consider the main components of the architecture.

Reference Monitor The standard function of a reference monitor is to disallow actions that are not authorized [52]. We augment this requirement so that the reference monitor also disallows controllable actions when the obligations that they cause to be incurred, or the action itself, would

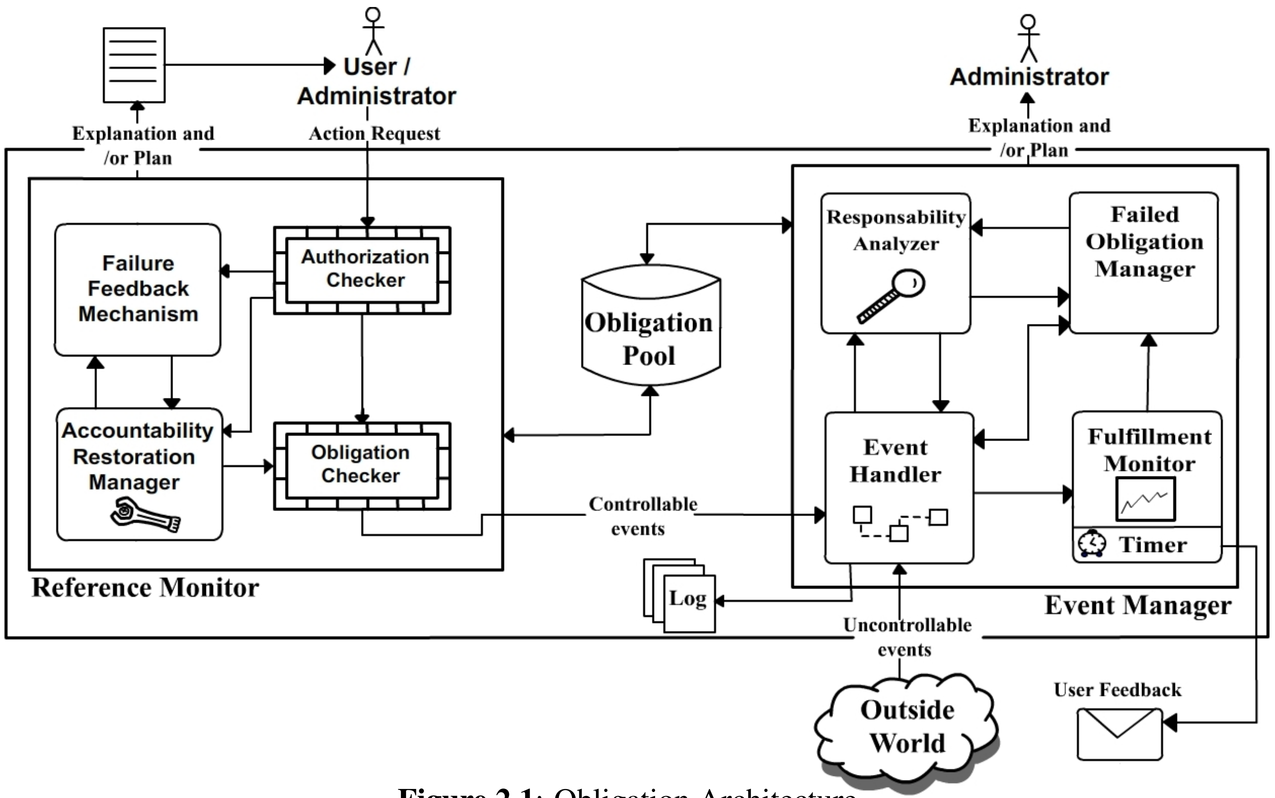

Figure 2.1: Obligation Architecture

violate the accountability property of the system. Requiring the reference monitor to check for violation of accountability is an attempt to maintain accountability as an invariant of the system. The Reference Monitor is further divided into four main components.

(a) **Authorization Checker** It checks standard authorization (*i.e.*, check whether someone has the proper authorization to execute an action).

(b) **Obligation Checker** It is responsible for denying actions that violate the accountability property of the system. The Obligation Checker can deny an action if one of the following cases occurs. (i) The action is administrative and can make an obligated user unable to perform her obligation. (ii) The action causes an obligation to be incurred that will not be authorized. (iii) The action introduces an administrative obligation that will make an obligated user unable to perform a subsequent obligation.

Denying an action that violates accountability is an attempt to maintain accountability incrementally as often as possible. By providing an algorithm for checking accountability and evaluating it empirically, it has been shown [45] that one can efficiently maintain accountability of a large scale system in practice. Please, consult chapter 3 for more details.

An additional function of this component is to select or help a user to select among the policy rules that can be used to authorized the desired action. When multiple rules preserve accountability, the appropriate rule to select may be application dependent. In some cases, it may even be appropriate to let the user requesting the action make the selection. When this is inappropriate, for example, due to performance issues, a range of policy-driven alternatives are possible. However, this is out of the scope of this dissertation

(c) **Failure Feedback Mechanism** When the Obligation Checker component denies an action because it would violate accountability, the failure feedback component attempts to present the user with an alternate plan of action that will enable her to accomplish her desired actions without violating accountability. The user has the option of accepting the plan, or not . If accepted, the actions in the plan become obligations. The plan can involve actions for the user herself and for others. If the plan contains actions for other users, they must also agree to the plan before the system will convert the plan into user obligations. If no plan is found, the action will be denied and the user will be notified. This problem is called the Action Failure Feedback Problem (*AFFP*). Pontual *et al.* [47] present a formal specification of the *AFFP*, complexity analysis, an AI-based approach that can encode instances of the AFFP problem as an input to a partial order AI planner, and empirical evaluation of the resulting tool. Please, consult chapter 4 for more details.

(d) **Accountability Restoration Manager** When an obligation is violated, unavailable, or some external uncontrollable event results in the violation of the system's accountability, this is detected by the failed obligation manager module (in the event manager) that will be discussed shortly in detail. When the administrator is notified of the violation, he uses the semi-automatic tool support provided by the Accountability Restoration Manager to restore accountability. For that, we consider that the obligation pool is an object that an administrator can edit. Restoring accountability is a complex problem. It requires consideration of characteristics of obligations such as their importance, purpose, and level of urgency. The capabilities of individual users that might be candidates for assuming obligations previously assigned to others must also be considered. Therefore, a fully operational, deployed obligation system must include human actors to handle or help with accountability restoration. Thus, the Accountability Restoration Manager includes a human, probably the same human as is generally responsible for obligation system management. Strategies for doing so are discussed in chapter 5.

Event Manager The major responsibilities of the event manager are altering authorization state of the system according to the administrative events, monitoring obligation status, and recognizing responsible users for obligation violation. The main components of it are presented next.

(a) **Event Handler** It observes controllable and uncontrollable events, and is also responsible for performing administrative events, modifying the authorization state accordingly. It also adds new obligations to the obligation pool, per the policy rule requirements. All the events are also logged by the event handler.

(b) **Fulfillment Monitor** It is responsible for checking whether an observed event constitutes the fulfillment of a pending obligation. It uses a timer to keep track of the obligations that are nearing their deadlines and notifies the appropriate users. Finally, it detects violated obligations and notifies the Failed Obligation Manager.

(c) **Failed Obligation Manager** It is responsible for determining the violated and unavailable obligations. After that, it determines all the obligations that are affected by them. (See chapter 5 section 5.3.)

(d) **Responsibility Analyzer** It is important for organizational managers to be aware of which employees are diligent and which are not. This component provides this information.

Assuming the system was initially in an accountable state, any single obligation violation is the responsibility of the user charged with fulfilling it. This also holds when multiple obligations are violated concurrently. When an administrative obligation is violated; other obligations may also be violated due to lack of permissions.

This component may not be executed/called immediately each time an obligation goes unfulfilled. Consequently, it is possible that this component is presented with multiple obligations that have been violated, some of which were supposed to be performed by users that had inadequate permissions. In this case, the user that is ultimately responsible for a violation is the one that had the permissions required to perform the administrative obligation to grant the missing permission. This module determines which users are responsible for causing each violation. Identifying users that are ultimately responsible for violations is known as the *blame assignment problem*, which is solved by this module.

Irwin *et al.* [32] provide a general approach for recognizing users responsible for accountability violation. However, they do not provide appropriate treatment to obligations incurred by unavailable users. We present techniques such as, reassignment of obligations, removal of obligations *etc.*, that an administrator can use to manage such obligations.

The architecture presented above is our vision for managing user obligations as part of the system's security policy. As mentioned at various points in the discussion above, several parts of this architecture (Obligation Checker [31, 45], Failure Feedback [47], Accountability Restoration Manager [46] and Responsibility Analyzer [32]) have been designed, implemented, and empirically evaluated. The remaining components are subjects of on-going and future work.

2.4 Abstract Model

This section presents an abstract meta-model that encompasses the basic constructs of an authorization system that supports obligations. Our presentation in this section is derived from that of Irwin *et al.* [31].

The abstract model presented here does not treat uncontrollable events and its effects on the system state. The majority of such events are going to cause only non-administrative obligations to be incurred. These can be handled by adding a policy rule that specifies the effect of such events as discussed below in this section. However, some events might cause a change in the authorization state, for instance, an employee getting arrested. In such cases, one can use the same techniques presented in chapter 5.

An obligation system consists of the following components:

- $\mathcal{U}$: a universe of users.
- $\mathcal{O}$: a universe of objects with $\mathcal{U} \subseteq \mathcal{O}$.
- $\mathcal{A}$: a finite set of actions that can be initiated by users. The structure of actions is given just below.
- $\mathcal{T}$: a countable set of time values.
- $\mathcal{B} = \mathcal{U} \times A \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$: the universe of obligations users can incur. Given $b = \langle u, a, \vec{o}, \text{start}, \text{end} \rangle \in \mathcal{B}$, $b.u$ denotes the obligated user, $b.a$ is the action the user must perform, $b.\vec{o}$ is the finite sequence of zero or more objects that are parameters to the action, and $b.\text{start}$ and $b.\text{end}$ are the start and end times of the interval during which the action must be performed[1]. A *well formed* obligation b satisfies $b.\mathrm{start} < b.\mathrm{end}$.

  User-initiated actions are events from the point of view of our system. We denote the universe of events that correspond to nonobligatory, discretionary actions by:

  $$\mathcal{D} = \mathcal{U} \times A \times \mathcal{O}^*$$

  We denote the universe of all controllable events, obligatory and discretionary, by $\mathcal{E} = \mathcal{D} \cup \mathcal{B}$.

- Γ : fully abstract representation of authorization state (*e.g.*, AC matrix, UA).
- $\mathcal{S} = \mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \mathcal{T} \times \Gamma \times \mathcal{FP}(\mathcal{B})$: the set of system states[2]. We use $s = \langle U, O, t, \gamma, B \rangle$ to denote system states. U is the finite set of users currently in the system, O is the finite set of objects, t is the current time, B is the set of pending obligations, and $\gamma \in \Gamma$.
- $\mathcal{P}$: a fixed set of policy rules. A policy rule $p \in \mathcal{P}$ has the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$, in which $a \in A$ (which means $\langle u, a, \vec{o} \rangle \in \mathcal{E}$) and $cond$ is a predicate that must be satisfied by $(u, \vec{o}, a)$ (denoted $\gamma \vDash cond(u, \vec{o}, a)$) in the current authorization state γ when

[1]Throughout, we refer to components of structured objects such as b with notation such as $b.\vec{o}$.
[2]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.

the rule is used to authorize the action. F_{obl} is an *obligation function*, which returns a finite set $B \subset \mathcal{B}$ of obligations incurred (by u or by others) when the action is performed under this rule.

Each action a denotes a higher order function of type $(\mathcal{U} \times \mathcal{O}^*) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma)$. When, in state s, user $u \in s.U$ performs action a on the objects $\vec{o} \in s.O^*$, $a(u,\vec{o})(s.U, s.O, s.\gamma)$ returns $\langle s'.U, s'.O, s'.\gamma \rangle$ for the new state s'. Thus, actions can introduce new users and objects, have side effects, and change the authorization state. Note that in general performing an action also introduces new obligations; these depend on the policy rule used, as well as on the action, and are handled in Definition 1 below.

Note that for a given action $e \in \mathcal{E}$ and a given policy rule $p \in \mathcal{P}$, the transition relation is deterministic. The following definition formalizes this relation.[3]

Definition 1 (Transition relation)**.** *Given any sequence of event/policy-rule pairs, $\langle e,p \rangle_{0..k}$, and any sequence of system states $s_{0..k+1}$, the relation $\longrightarrow \subseteq \mathcal{S} \times (\mathcal{E} \times \mathcal{P})^+ \times \mathcal{S}$ is defined inductively on $k \in \mathbb{N}$ as follows:*

(1) We have $s_k \overset{\langle e,p \rangle_k}{\longrightarrow} s_{k+1}$ if and only if the policy rule $p_k = a(u,\vec{o}) \leftarrow cond(u,\vec{o},a) : F_{obl}(s,u,\vec{o}) \in \mathcal{P}$ satisfies $a = e_k.a$, and $s_{k+1} = \langle U'', O'', t'', \gamma'', B'' \rangle$ satisfies

$$s_k.\gamma \vDash cond(u,\vec{o},a)$$

$$e_k.u \in s_k.U \text{ and } e_k.\vec{o} \in s_k.O^*$$

$$\langle U'', O'', \gamma'' \rangle = a(u,\vec{o})(s_k.U, s_k.O, s_k.\gamma)$$

$$B'' = \begin{cases} (s_k.B - \{e\}), \text{ if } e_k \in \mathcal{B} \\ s_k.B \cup F_{obl}(s_k, e_k.u, e_k.\vec{o}), \text{ otherwise} \end{cases}$$

and

[3]Notation: for $j \in \mathbb{N}$, we use $s_{0..j}$ to denote the sequence $s_0, s_1, \ldots, s_j$, and for $\ell \in \mathbb{N}$, $\ell \leq j$, $s_{0..\ell}$ denotes the prefix of $s_{0..j}$ and when $\ell < j$ the prefix is *proper*. Similarly, $\langle e,p \rangle_{0..j}$ denotes $\langle e_0,p_0 \rangle, \langle e_1,p_1 \rangle, \ldots, \langle e_j,p_j \rangle$.

(2) $s_0 \xrightarrow{\langle e,p\rangle_{0..k}} s_{k+1}$ *if and only if there exists* $s_k \in \mathcal{S}$ *such that* $s_0 \xrightarrow{\langle e,p\rangle_{0..k-1}} s_k$ *and* $s_k \xrightarrow{\langle e,p\rangle_k} s_{k+1}$.

2.5 Concrete Model

This section presents two instantiations of the above abstract obligation system that will be used on the reminder of this dissertation. Section 2.5.1 presents the definitions of the mini-RBAC/mini-ARBAC authorization model, as well, the obligation concrete model instantiated with it, and some examples. In section 2.5.2 we present the formalization of the extended mini-HRU Access Control Matrix Model, the instantiation of our obligation model using it as the authorization model. This section also presents some examples.

2.5.1 mini-RBAC and mini-ARBAC Authorization Model

Role Based Access Control (RBAC) unlike AC matrix model uses a level of indirection between users and permissions called “roles”. Instead of assigning permissions to users directly, RBAC allows assigning permissions to roles and roles to users. Introducing roles makes it easier to manage users and their associated permissions in a large organization.

Administrative Role Based Access Control (ARBAC) model is used to manage RBAC. ARBAC allows administrators to assign and remove roles from users. The widely studied RBAC96 [49] and ARBAC97 model [50] have been simplified somewhat by Sasturkar *et al.* for the purpose of studying policy analysis, forming a family of languages called *mini-RBAC* and *mini-ARBAC* [51]. The member of the family that we use supports administrative actions that modify user-role assignments, but does not consider explicit separation of duty (SMER), role hierarchies (however, a policy with role hierarchy can be easily transformed in one that does not support it), sessions, changes to permission-role assignments, or role administration operations, such as creation or deletion of roles. The model does not distinguish between regular and administrative roles. The presentation of mini-RBAC and mini-ARBAC here is based on the one given by Sasturkar *et al.* [51]. We present examples of a mini-RBAC policy in table 2.1, and a mini-ARBAC policy in table 2.2 on page 26.

We use mini-ARBAC and mini-RBAC in this study in part because of its relationship with RBAC, an access control model that has gained wide acceptance in many sectors. Previous work [31] in accountability analyzed obligations in the context of the AC matrix model. It is interesting and useful to investigate issues that arise in integrating the user obligation systems with another popular authorization system. It turns out that mini-ARBAC and mini-RBAC are simpler than the AC matrix model, because administrative operations are more restricted. In the AC matrix model, individual administrative actions can make multiple changes to the access control state, whereas in mini-ARBAC, actions can change only one permission or role assignment at a time. Moreover, the impact of those changes are simpler, as authorization depends on individual role memberships, rather than on a combination of matrix entries. As we shall see, this simplification has a beneficial impact on the complexity of deciding strong accountability.

Definition 2 (mini-RBAC:)**.** *A mini-RBAC authorization state* γ *is a tuple* $\langle U, R, P, UA, PA\rangle$ *where:*

- U, R and P are the finite sets of users, roles and permissions respectively, where a permission $\rho \in P$ is a pair $\langle action, object\rangle$.
- $UA \subseteq U \times R$ is a set of user-role pairs. Each $\langle u, r\rangle \in UA$ indicates that user u is a member of role r.
- $PA \subseteq R \times P$ is a set of permission-role pairs. If $\langle r, \rho\rangle \in PA$, users in role r are granted the permission ρ.

Definition 3 (mini-ARBAC policy:)**.** *A mini-ARBAC model* ψ *is a tuple* $\langle CA, CR\rangle$ *where:*

- $CA \subseteq R \times C \times R$ is a set of the *can_assign* rules, in which C is the set of preconditions, the structure of which is discussed presently. Each $\langle r_a, c, r_t\rangle \in CA$ indicates that users in role r_a are authorized to assign a user to the *target role* r_t, provided the target user's current role memberships satisfy precondition c. A precondition is a conjunction of positive and negative roles. Target user u_t satisfies c in γ (written $u_t \vDash_\gamma c$) if for each literal l in c, $u_t \vDash_\gamma l$, which is defined by $u \vDash_\gamma r \equiv \langle u, r\rangle \in \gamma.UA$ and $u \vDash_\gamma \neg r \equiv \langle u, r\rangle \notin \gamma.UA$. So the action *grant*($u_a$,$r_t$,$u_t$) is authorized if there exists $\langle r_a, c, r_t\rangle \in CA$ such that $u_a \vDash_\gamma r_a$ and $u_t \vDash_\gamma c$.

- $CR \subseteq R \times R$ is a set of *can_revoke* rules. Each $\langle r_a, r_t\rangle \in CR$ indicates that a user belonging to the (administrative) role r_a has the capability to revoke the role r_t from any target user (*i.e.*, there exists $\langle r_a, r_t\rangle \in CR$ such that $u_a \vDash_\gamma r_a$). There are no constraints on revocation based on the other roles held by the target user.

Table 2.1: An example mini-RBAC authorization state, γ, for a software development life cycle. Note that we allow the use of wildcards (*) in the expression of permissions to denote collections of permission pairs.

U	=	{Joan, Carl, Alice, Bob, Eve}
R	=	{projectManager, developer, blackBoxTester, securityManager}
P	=	{⟨develop, sourceCode⟩, ⟨test, software⟩, ⟨assignProjObl, * ⟩}
UA	=	{⟨Joan, securityManager⟩, ⟨Alice, developer⟩, ⟨Bob, blackBoxTester⟩, ⟨Eve, projectManager⟩}
PA	=	{⟨developer, ⟨develop, sourceCode⟩⟩, ⟨projectManager, ⟨assignProjObl, *⟩⟩, ⟨blackBoxTester, ⟨test, software⟩⟩}

Table 2.2: An example mini-ARBAC policy, ψ, for a software development life cycle.

CA	=	{⟨securityManager, ¬ blackBoxTester, developer⟩, ⟨securityManager, ¬ developer, blackBoxTester⟩}
CR	=	{⟨securityManager, blackBoxTester⟩}

mini-RBAC/mini-ARBAC Concrete Model

In the instantiated concrete model we introduce here, $\mathcal{O}$ is a set of objects with $\mathcal{U} \cup R \subseteq \mathcal{O}$, in which R is the set of roles and $\mathcal{U}$ is the set of users of the system. The finite set of actions A comprises of two different types of actions, administrative actions (*e.g.*, *grant* and *revoke*) and non-administrative actions (*e.g.*, *read*, *write*, *etc.*). An event e is a tuple $\langle u, a, \vec{o}\rangle$ in which u is the user performing the action, a is the action (*i.e.*, *grant*, *revoke*, *read*, *etc.*), $\vec{o}$ is a tuple of objects, the subtype of which depends on the action. For instance, $\vec{o} = \langle r_t, u_t\rangle$ when a is an administrative action, and $\vec{o}$ may be $\langle book\rangle$ when a is the non-administrative action (*e.g.*, *read*).

In the concrete model, we use a mini-RBAC policy $\gamma = \langle P, R, UA, PA, \rangle$ and a mini-ARBAC $\psi = \langle CA, CR \rangle$. Often left fully abstract in RBAC models, permissions $P \subset A \times \mathcal{O}^*$ in our context is taken to be a set of action, object-tuple pairs. We omit U because it occurs elsewhere in the obligation-system state s. In our context, the authorization state γ can be modified dynamically according to the policy ψ. As in the abstract model, the set of obligation-system policy rules $\mathcal{P}$ consists of policy rules of the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. The way in which it is determined whether the condition $cond(u, \vec{o}, a)$ is satisfied in a given authorization state γ depends on whether a is an administrative action. When a is grant (respectively, revoke), $\vec{o}$ is a pair $\langle u_t, r_t \rangle$ and u is attempting to grant role r_t to (respectively, revoke r_t from) user u_t. In this case u's authorization to do so is determined by $\gamma.UA$ and $\psi.CA$ (respectively, $\psi.CR$). When action a is not administrative, u's authorization is determined by $\gamma.UA$ and $\gamma.PA$.

Let us now consider the transition from state s to some state s' that occurs when an event e is handled according to policy rule p. The fact that this is the transition taken is denoted by $s \xrightarrow{\langle e,p \rangle} s'$. Letting $e = (u, a, \vec{o})$, we require that $u \in s.U$ and $\vec{o} \in s.O^*$. The action $a(u, \vec{o})$ determines the values $\langle s'.U, s'.O, s'.\gamma \rangle$ based on $(s.U, s.O, s.\gamma)$. Thus, actions can introduce and remove users and objects and change the authorization state. The condition, $cond(u, \vec{o}, a)$, in the policy rule p must be satisfied for p to be used in the transition; F_{obl} associated with the particular p determines any new obligations added in obtaining $s'.B$ from $s.B$. These points are formalized in definition 5 below.

There are three cases in which a user u is authorized to perform an action a on an object tuple $\vec{o}$. (i) When a is non-administrative, authorization depends on the permissions assigned to u's roles; (ii) when a grants a role, there must be a *can_assign* rule for one of u's roles such that the target user u_t satisfies the precondition; (iii) when a revokes a role, a similar requirement holds on the existence of a *can_revoke* rule. This is formalized in definition 4.

Definition 4. *For all* $u \in \mathcal{U}$ *and* $\vec{o} \in \mathcal{O}^*$, $\gamma \vDash cond(u, \vec{o}, a)$ *if and only if the following holds*[4].

$$
\begin{aligned}
&(\exists r).(((u, r) \in \gamma.UA) \wedge \\
(i) \quad & [a \notin \text{administrative} \rightarrow (\langle r, \langle a, \vec{o} \rangle \rangle \in \gamma.PA)] \wedge \\
(ii) \quad & (\forall u_t, r_t).[a = grant \wedge \vec{o} = \langle u_t, r_t \rangle \rightarrow \\
& \qquad (\exists c).((\langle r, c, r_t \rangle \in \psi.CA) \wedge (u_t \vDash_\gamma c))] \wedge \\
(iii) \quad & (\forall u_t, r_t).[a = revoke \wedge \vec{o} = \langle u_t, r_t \rangle \rightarrow \\
& \qquad ((\langle r, r_t \rangle \in \psi.CR))])
\end{aligned}
$$

The transition relation presented in definition 1 preserves the invariant over states $s = \langle U, O, t, \gamma, B \rangle$ given by $\forall b \in s.B \cdot (b.u \in s.U) \wedge (b.o^* \subseteq \mathcal{O}^*)$ and $s.U = s.\gamma.U$.

Definition 5 (Transition relation)**.** *Given any sequence of event/policy-rule pairs,* $\langle e, p \rangle_{0..k}$, *and any sequence of system states* $s_{0..k+1}$, *the relation* $\longrightarrow \subseteq \mathcal{S} \times (\mathcal{E} \times \mathcal{P})^+ \times \mathcal{S}$ *is defined inductively on* $k \in \mathbb{N}$ *as follows:*
(1) $s_k \overset{\langle e,p \rangle_k}{\longrightarrow} s_{k+1}$ *holds if and only if, letting* $p_k = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$, *we have* $s_k.\gamma \vDash cond(e_k.u,\ e_k.\vec{o},\ e_k.a)$, *and* $s_{k+1} = \langle U'', O'', t'', \gamma'', B'' \rangle$, *in which* $\langle U'', O'', \gamma'' \rangle = a(u, \vec{o})(s_k.U, s_k.O, s_k.\gamma)$, $B'' = (s_k.B - \{e_k\})$ *when* $e_k \in \mathcal{B}$, *and* $B'' = s_k.B \cup F_{obl}(s_k, e_k.u, e_k.\vec{o})$ *otherwise.*
(2) $s_0 \overset{\langle e,p \rangle_{0..k}}{\longrightarrow} s_{k+1}$ *if and only if there exists* $s_k \in \mathcal{S}$ *such that* $s_0 \overset{\langle e,p \rangle_{0..k-1}}{\longrightarrow} s_k$ *and* $s_k \overset{\langle e,p \rangle_k}{\longrightarrow} s_{k+1}$.

Note that a reference monitor in our obligation system is going to require more than simply that a transition is well defined to permit an action to be performed according to a given policy rule. It will further require that performing the action and adding any obligations required by the policy rule leaves the system in an accountable state.

Example 6 (Obligation System)**.** *We use the mini-RBAC, and mini-ARBAC policies presented in tables 2.1 and 2.2 respectively to illustrate three scenarios that demonstrate how an obligation*

[4]We use "[" and "]" as an alternate form of parenthesis to aid the eye in recognizing the formula's syntactic structure

system can be used to manage a software development cycle. For simplicity, we assume all the roles and permissions in this example are associated with a specific software development project.

In scenario 1, Bob has an obligation to perform black-box testing of some software. Should the security manager, Joan, attempt to revoke Bob's black-box tester role, she would be prevented from doing so. This is because Bob needs the role to fulfill his obligation, so revoking it would make the system unaccountable. (Of course, in some situations, such as Bob leaving the company, Joan would have to be able to force revocation. This requires handling the violation of accountability, in chapter 4 we present techniques to restore accountability.)

In scenarios 2 and 3, Eve, the project manager performs discretionary actions that assign obligations to team members. For this, she uses the action $assignProjObl$, which is governed by the following policy rule in our framework:

$$\begin{aligned}&\mathit{assignProjObl}(\mathit{pm}, \langle \mathit{oblAction}, \mathit{oblUser}, \overrightarrow{\mathit{oblObject}}, \mathit{oblStart}, \mathit{oblEnd}\rangle)\\&\leftarrow (\mathit{pm} \vDash_\gamma \mathit{projectManager}) : \{\langle \mathit{oblUser}, \mathit{oblAction}, \overrightarrow{\mathit{oblObject}}, [\mathit{oblStart}, \mathit{oblEnd}]\rangle\}\end{aligned}$$

In scenario 2, Eve creates a new obligation that requires Alice to perform black-box testing within 1 month. The action Eve performs is given by $\mathit{assignProjObl}(\mathit{Eve}, \langle \mathit{test}, \mathit{Alice}, \langle \mathit{software}\rangle, 01/01/2010, 02/01/2010\rangle$. Eve satisfies $\mathit{Eve} \vDash_\gamma \mathit{projectManager}$, so the authorization system permits her to perform the action. However, the new obligation, $\langle \mathit{Alice}, \mathit{test}, \langle \mathit{software}\rangle, [01/01/2010, 02/01/2010]\rangle$, would make the system unaccountable, since Alice does not have the role of black-box tester. So Eve's action is prevented.

In scenario 3, after discovering that Alice does not have the black-box tester role, Eve attempts to create a new obligation that obligates Joan to grant Alice the role of black-box tester. For this, Eve attempts the action, $\mathit{assignProjObl}(\mathit{Eve}, \langle \mathit{Grant}, \mathit{Joan}, \langle \mathit{blackBoxTester}, \mathit{Alice}\rangle, 01/01/2010, 02/01/2010\rangle)$. This would generate the new obligation $\langle \mathit{Joan}, \mathit{Grant}, \langle \mathit{blackBoxTester}, \mathit{Alice}\rangle, [01/01/2010, 02/01/2010]\rangle$. However, as Alice does not satisfy the conditions required for assignment to this role, Joan would be unable to fulfill this obligation.

2.5.2 Extended mini-HRU Access Control Matrix Model

The HRU Access Control Matrix Model [27] is a well-known and popular access control model which is expressive enough to model many real world systems. However, we extend the original HRU Access Control Matrix Model in a way that conditional commands can have positive and negative conditions, and the conditions are expressed in a disjunctive normal form (DNF). In addition, it also supports non-administrative commands (*i.e.*, commands that do not modify the matrix of rights). In contrast to the original HRU model, we only consider two primitive administrative operations, namely, *remove a right* and *enter a right*. The components that comprise the extended mini-HRU model is discussed below:

Definition 7 (Extended mini-HRU Access Control Matrix Model:). *An HRU Access Control Matrix Model γ is a tuple $\langle \mathcal{R}, \mathcal{O}, \mathcal{U}, M, \mathcal{AC}, \mathcal{NA} \rangle$ where:*

- $\mathcal{R}$, $\mathcal{O}$ and $\mathcal{U}$ are respectively the finite set of rights, objects and users.
- $M = \mathcal{FP}(\mathcal{U} \times \mathcal{O} \times \mathcal{R})$ is the matrix of rights, which represents the set of finite permissions. Individual positive permissions are given by $m = (u, o, r)$ when $r \in M[u, o]$, and negative permission are denoted by $\neg m = (u, o, \neg r)$ when $r \notin M[u, o]$. This signifies that a user u has a right r over an object o when $r \in M[u, o]$, and u does not have right r over object o when $r \notin M[u, o]$.
- $\mathcal{AC}$ is a finite set of administrative commands where each command performs a finite sequence of primitive administrative operations. Each primitive administrative operation can enter (resp., remove) a right to (resp., from) into M. Thus, individual administrative commands can make multiple changes to M.

  Each of the administrative command $ac \in \mathcal{AC}$ has the following form:

```
command $\alpha(X_1, X_2, ..., X_k)$
 if $c$ then
    $op_1, ..., op_n$
```

end

α represents the name of the command, $X_1, ..., X_k$ are the formal parameters to the command (*i.e.*, users and/or objects). A precondition c is a conjunction of positive and negative permissions. A command can be executed only if the precondition is evaluated to be true according to M. We say that a precondition $c = \bigwedge_i l_i$ is satisfied when all of its conjuncts $l_i \in c$ are satisfied with respect to M denoted by $l_i \vDash M$.

$$l_i \vDash M = \begin{cases} r \in M[u,o] & \text{if } l_i = m_i \text{ where } m_i = (u,o,r) \\ r \notin M[u,o] & \text{if } l_i = \neg m_i \text{ where } m_i = (u,o,r) \\ \text{false} & \text{otherwise} \end{cases}$$

In contrast to the original HRU Access Control Matrix Model, our extended model supports multiple commands with the same name α. When this happens, the commands differ only on their preconditions, in such a way that is equivalent to having a unique command with a precondition expressed as a disjunction of the separate preconditions[5]. Each op_i in the command represents a primitive operation that can alter M. The extended mini-HRU Access Control Matrix Model supports the following two primitive operations:

- **enter** r **into** $M[u,o]$: This operation adds the right r to the cell $M[u,o]$.
- **remove** r **from** $M[u,o]$: This operation removes the right r from the cell $M[u,o]$.

When a command is executed, the effect of the primitive operations associated with it is atomic. Let us consider the following example.

Example 8 (Atomicity)**.** *In this example we demonstrate that the effect of our administrative commands are atomic.*

```
command $grant \bullet r_t(u_a, u_t, o_t)$
    if $(own \in M[u_a, o_t]) \wedge (r_x \in M[u_t, o_t])$ then
```

[5]Due to the restriction of the model, the resulting precondition would be in disjunctive normal form.

```
    $M[u_t, o_t] := M[u_t, o_t] \setminus \{r_x\}$
    $M[u_t, o_t] := M[u_t, o_t] \cup \{r_t\}$
    $M[u_t, o_t] := M[u_t, o_t] \cup \{r_x\}$
  end
```

In the previous command, we can see that the first primitive operation removes the r_x right from $M[u_t, o_t]$. However, the last primitive operation restores the r_x right to $M[u_t, o_t]$. Due to the atomic nature of the command, it seems like the right r_x was never removed. For this reason, we just consider the end effect of the command. In this example, the end effect of executing this command is user u_t will end up with the rights r_x and r_t over object o_t.

- $\mathcal{NA}$ represents the set of non-administrative commands. Recall that the original HRU Access Control Matrix Model does not support non-administrative commands. Each of the non-administrative commands $na \in \mathcal{NA}$ has the following form:

  ```
  command $\alpha(X_1, X_2, ..., X_k)$
   if $c$ then $\alpha$ is authorized
  end
  ```

  The terminology used in non-administrative commands is similar to that of administrative commands. Note that, we could use an administrative command to simulate a non-administrative command. This can be trivially accomplished by entering a right that appears in the condition of the command, (*e.g.*, if we have a command that checks $read \in M[u, o]$, then we could have the following primitive $M[u, o] \cup \{read\}$. Note that, this primitive does not alter the cell $M[u, o]$).

  Example 9 (Non-administrative command)**.** *In this example we present a non-administrative command that allows a user to read a file.*

  ```
  command $read \bullet file(u, file)$
    if $(read \in M[u, file])$ then
      read file
  ```

```
    end
```

Extended mini-HRU Access Control Matrix Concrete Model

We instantiate our model to use HRU Access Control Matrix Model as the authorization state. In our instantiated concrete model $\mathcal{O}$ is a set of objects with $\mathcal{U} \subseteq \mathcal{O}$, in which $\mathcal{U}$ is the set of users of the system. The finite set of actions $\mathcal{A}$ comprises of two different types of actions, administrative actions (*e.g.*, *grant a right* and *revoke a right*) and non-administrative actions (*e.g.*, *read*, *write*, *etc.*). Each action $a \in \mathcal{A}$ is now assumed to perform a command from the set $\mathcal{AC} \cup \mathcal{NA}$. When a performs a command from the set $\mathcal{AC}$ (resp., $\mathcal{NA}$) it is called an administrative (resp., a non-administrative) action. Note that, for the rest of the dissertation, actions and commands will mean the same thing and will be used interchangeably.

Example 10 (Administrative Obligation)**.** *In this example we present an administrative obligation* $b_1 = \langle grant \bullet r_t(u_a, u_t, o_t), [5, 10]\rangle$ *that utilizes the following command.*

```
command $grant \bullet r_t(u_a, u_t, o_t)$
 if $(own \in M[u_a, o_t]) \wedge (r_1 \in M[u_t, o_3])$ then
   $M[u_t, o_t] = M[u_t, o_t] \cup \{r_t\}$
 end
```

In our context, the rights r in M are modified dynamically when some administrative action takes place. The set of obligation-system policy rules $\mathcal{P}$ consists of policy rules of the form $a(st, u, O) \leftarrow cond : F_{obl}$. The user u will be authorized to perform action a on object O in state st if it satisfies $cond(u, t, \sigma, O)$ associated with that action. The condition $cond(u, t, \sigma, O)$ is satisfied when the user u is authorized to perform action a on the set of objects O at time t according $\gamma.M$. Here, $\sigma = \gamma$, where γ is a HRU Access Control Matrix Model. The $cond(u, t, \sigma, O)$ of an action a gives the pre-condition for the command associated with a.

2.6 Summary

This chapter has presented background necessary to understand the contributions of this dissertation. It also has given a brief overview of each of the components of our obligation framework and discuss their interactions. A central organizing principle of the architecture is that the system should be in an accountable state as much of the time as possible without interfering unnecessarily with usability. Furthermore, we have also presented the abstract obligation model. It has also reviewed the mini-ARBAC/mini-ARBAC and the extended mini-HRU authorization models. It then have defined the obligation concrete models instantiated with both authorization models.The next chapters use the concrete obligation models presented in this chapter.

CHAPTER 3: ACCOUNTABILITY PROPERTIES

This chapter presents a reformulation of the definitions of strong and weak accountability properties [31]. Accountability properties are defined in terms of hypothetical schedules according to which the given pool of obligations could be executed, starting in a given state. Strong accountability requires that each obligation be authorized throughout its entire time interval, no matter when during their intervals the other obligations are scheduled, and no matter which policy rules are used to authorize them. On the other hand, weak accountability allows an obligation to be unauthorized during part of its time interval, provided that if the obligated user waits for other obligations to be fulfilled, it is guaranteed that the action will become authorized before its deadline.

The property's name, accountability, reflects the fact that it is appropriate for an organization to hold those accountable who fail to perform authorized obligations. To the extent that it can be maintained, accountability is helpful for heightening the benefits derived by assigning user obligations. These benefits include facilitating effective planning, including obtaining early warning when plans are infeasible, as well as transparency and awareness of which users are failing to fulfill their duties and the impact of such failures.

Deciding whether a system state is accountable requires reasoning about future states of the authorization system. It is reasonable to assign to a user an obligation that the user is not currently authorized to perform, provided that other obligations will grant that authorization prior to the time the first obligation must be performed. As we have discussed in chapter 2, a reference monitor can help maintain accountability by preventing actions that would cause it to be violated. To this end, it is necessary to study how accountability checking can impact the performance of the reference monitor.

Recall that the results presented by Irwin *et al.* were inconclusive from the standpoint of practicality. When they instantiated their obligation model with a simplified version of the HRU Access Control Matrix Model [27], the problem of determining strong accountability becomes polynomial. However for the algorithm they present, this polynomial has degree 4 in the number of obligations and degree 2 in the policy size. Furthermore, no performance evaluation is provided to determine the limits of problem instance size that can be handled efficiently in practice. Additionally, determining weak accountability in the instantiated case remains intractable (co-NP hard) and no decision procedures are presented.

In this chapter, we instantiate the authorization portion of the obligation model of Irwin *et al.* with a previously studied administrative role-based access control model [48, 50] called mini-RBAC and mini-ARBAC [51]. Using mini-ARBAC and mini-RBAC instead of the extended mini-HRU Access Control Matrix Model simplifies the problem of determining accountability, largely because obligatory actions are limited to making at most one change to the authorization state per action. Our treatment here ignores the possibility of "cascading" of obligations. Specifically, we achieve this by assuming actions that can cause new obligations to be incurred are disjoint from actions that can be obligatory (in chapter 6, we generalize the definition of the strong accountability property considering arbitrary cascading of obligations and also present decision procedures of strong accountability that can handle with restricted form of cascading obligations). We also present decision procedures for strong and weak accountability, which we then evaluate empirically. Finally, we present an improved decision procedures for strong accountability when considering the extended mini-HRU Access Control Matrix Model presented in previous chapter as the authorization state. Empirical evaluations of these procedures are also presented.

The results presented on this chapter support the thesis that maintaining strong accountability in the reference monitor is reasonable in most applications, and that in many, even weak accountability can be supported adequately.

3.1 Illustrations and Utility of Accountability

Before we formally define accountability properties, let us present four simple examples in a software-development environment that illustrate the use of accountability in detecting that obligatory actions are unauthorized when the obligation is introduced, rather than when the action is attempted. Let us assume Eve is a project manager. She uses the action *assignProjObl* to assign obligations to team members. The action *assignProjObl* takes as input the values that are placed into the new obligation.

Suppose that Alice's only role is that of a developer, which enables her to develop software. In scenario 1, Eve creates an obligation that requires Alice to perform black-box testing. In doing so, Eve makes the state unaccountable, as Alice does not have the requisite roles to perform this action. Thus, a reference monitor that tries to enforce accountability will prevent Eve from performing this *assignProjObl*. Without this intervention, the inadequacy of Alice's roles would be discovered only when Alice attempts to perform the testing.

Now, suppose Bob has the role blackBoxTester. Assume that the organization uses mandatory vacation to help prevent insiders committing fraud [9] and that Eve is responsible for ensuring her employees adherence to this policy. In scenario 2, Bob is required to go a mandatory vacation in July. Eve adds obligations that Joan, in the role securityManager, removes Bob's roles while Bob is on vacation and restore them when he returns. Now, if Paul, another projectManager, tries to assign to Bob an obligation to perform black-box testing in July, Paul will be prevented from doing so, as Bob will not have the necessary roles at that time. Paul discovers this when he tries to assign obligations to Bob rather than when Bob attempts to perform the task or fails to make the attempt due to being out of the office.

In scenario 3, suppose Bob already has an obligation to perform black-box testing of a software component. Should Joan attempt to revoke Bob's blackBoxTester role, she would be prevented because Bob will need it to fulfill his existing obligation. Under normal circumstances, this prevents Joan inadvertently removing a needed role. However, in some situations, such as when Bob leaves the company, Joan must be able to force the role revocation and in this case must remove or reassign Bob's obligations. Either of these courses of actions could interact with other obligations already in the pool. For instance, there might not be another user to whom Bob's obligation can be assigned. Simply removing the obligation might be unacceptable, either because the action is required in its own right or because some later obligation depends on its having been performed. Sometimes there will be no satisfactory solution, as when a key employee leaves the company.

However, to assist Joan in managing the situation when a solution exists, it is necessary to provide tools that support finding such a solution.

Assume the company does not allow a developer to perform black-box testing (i.e., a separation of duty policy). In scenario 4, Eve wants Alice to perform some black-box testing. To this end, Eve attempts to create a new obligation to Alice perform black-box testing. Her action will be denied, and she is going to discover that Alice does not have the blackBoxTester role. After discovering that, Eve tries to add a new obligation to Joan to grant Alice the role of black-box tester. However, as Alice does not satisfy the conditions required for assignment to this role, Eve again will have her action denied. Note that in these scenarios, the system alerts Eve immediately that the tasks she is attempting to assign cannot be completed, and enables her to revise her plans accordingly. Without the accountability checks, Eve would not be alerted to the fact that the obligations cannot be fulfilled until the obligated users attempted the obligatory actions and were prevented from performing them by the authorization system. The above scenarios illustrate the utility of accountability when considering obligations that have only authorization dependencies among themselves. Note that, it is possible to extend our notion of accountability to support other notions of dependencies, for instance, functional dependencies. For example, Bob has an obligation b_1 to create a report in June, and Alice has an obligation b_2 to review the report in July. Obligation b_2 has a functional dependency on obligation b_1. If obligation b_1 does not happen, obligation b_2 cannot happen. However, addressing these is a matter of future work.

3.2 Strong Accountability Property

As discussed in the introduction, even though users are capable of failing to fulfill their obligations, it is very helpful to make use of a conditional notion of correctness that says, roughly, assuming the users fulfill their obligations diligently, all users will be authorized to perform their obligations in the appropriate time interval. This section formalizes this notion called strong accountability Accountability is defined in terms of hypothetical schedules according to which the given pool of obligations could be executed, starting in a given system state. Under the assumption that each prior obligation has been fulfilled during its specified time interval, accountability requires that each obligation be authorized throughout its entire time interval, no matter when during that interval the other obligations are scheduled, and no matter which policy rules are used to authorize them.

Given a set of obligations B, a *schedule* of B is a sequence $b_{1..n}$ that enumerates B, in which $n = |B|$ (*i.e.*, $|B|$ returns the size of the set B). A schedule of B is *valid* if for all i and j, if $1 \leq i < j \leq n$, then b_i.start $\leq b_j$.end. This prevents scheduling b_i before b_j if b_j.end $< b_i$.start. Given a system state s_1, and a policy $\mathcal{P}$, a proper prefix [1] $b_{1..j}$ of a schedule $b_{1..n}$ for B is *authorized by* policy-rule sequence $p_{1..j} \subseteq \mathcal{P}^*$ if there exists s_{j+1} such that $s_1 \overset{\langle b,p\rangle_{1..j}}{\longrightarrow} s_{j+1}$.

Let us now formalize the strong accountability property.

Definition 11 (Strong accountability)**.** *Given a state $s_1 \in \mathcal{S}$ and a policy $\mathcal{P}$, we say that s_1 is* strongly accountable *(denoted by $sa(s_1, \mathcal{P})$) if for every valid schedule, $b_{1..n}$, every proper prefix of it, $b_{1..k}$, for every policy-rule sequence $p_{1..k} \subseteq \mathcal{P}^*$ and every state s_{k+1} such that $s_1 \overset{\langle b,p\rangle_{1..k}}{\longrightarrow} s_{k+1}$, there exists a policy rule p_{k+1} and a state s_{k+2} such that $s_{k+1} \overset{\langle b,p\rangle_{k+1}}{\longrightarrow} s_{k+2}$.*

Example 12 (Strong Accountability)**.** *Using the mini-RBAC, mini-ARBAC policies presented in chapter 2 tables 2.1 and 2.2, let us assume we have two pending obligations b_1 and b_2 defined as follows:*

- b_1 : *Joan must grant the developer role to Carl in time* $[7, 9]$
- b_2 : *Carl must develop sourceCode in time* $[10, 20]$

The example obligation system is strongly accountable because obligation b_2 is guaranteed to be authorized between time 10 to 20, as obligation b_1 can be fulfilled any time in its time interval, since Joan already has the role of security manager and Carl satisfies the CA rule constraints.

[1]Notation: for $j \in \mathbb{N}$, we use $s_{1..j}$ to denote the sequence $s_1, s_2, \ldots, s_j$, and for $\ell \in \mathbb{N}$, $\ell \leq j$, $s_{1..\ell}$ denotes the prefix of $s_{1..j}$ and when $\ell < j$ the prefix is *proper*. Similarly, $\langle e,p\rangle_{1..j}$ denotes $\langle e_1,p_1\rangle, \langle e_2,p_2\rangle, \ldots, \langle e_j,p_j\rangle$.

3.2.1 Algorithm for Determining Strong Accountability

This section begins by presenting our incremental algorithm for determining whether adding a new obligation to a strongly accountable obligation pool preserves the property. It then discusses the complexity of the algorithm. A non-incremental version of the algorithm is then discussed. The algorithms presented here and in the next section are specialized to mini-ARBAC and mini-ARBAC, but can be generalized to support other models in which user rights are modified by administrative actions. As discussed in the section above, all the techniques presented here make the assumption that obligatory actions cannot cause new obligations to be incurred.

3.2.2 The Algorithm

Algorithm 1 is designed to be used incrementally for maintaining strong accountability. Algorithm 1 takes as input a strongly accountable set of obligations B, a mini-RBAC authorization state γ, a mini-ARBAC policy ψ, and a new obligation b. It returns *true* if adding b to B preserves strong accountability, and *false* otherwise. Below, we discuss using it to determine whether a given set of obligations is strongly accountable.

Algorithm 1 *StrongAccountable* (γ, ψ, B, b)

Input: A policy $\langle\gamma, \psi\rangle$, a strongly accountable obligation set B, and a new obligation b.
Output: returns **true** if addition of b to the system preserves strong accountability.

```
1: if Authorized ($\gamma$, $\psi$, $B$, $b$)= false then
2:     return false
3: if $b.a \neq$ grant or revoke then
4:     return true
5: After:={$b'|b' \in B \wedge b'$.end> $b$.start}
6: $B := B \cup b$
7: for each obligation $b^* \in$After do
8:     if Authorized ($\gamma$, $\psi$, $B$ , $b^*$)=false then
9:         $B := B \setminus b$ /* Restore representation of $B$ */
10:        return false
11: return true
```

The intuition behind Algorithm 1 is as follows. To determine whether adding b to B preserves strong accountability, the algorithm inspects the current authorization state $\langle\gamma, \psi\rangle$ and each obligation $b' \in B$ that could be performed prior to b ($b' \in B \wedge b'.\mathrm{end} \leq b.\mathrm{start}$) to determine whether

b will be authorized during its entire time interval, [b.start, b.end]. The algorithm uses procedure *Authorized* (Algorithm 2) for this purpose. If $b.a$ is an administrative action, the algorithm also determines whether b interferes with authorizations required by later obligations b^* ($b^* \in B \wedge b^*$.end $>$ b.start). When $b.a$ is not administrative, this is not necessary, as it does not affect authorizations.

Because $b.u$'s roles can change during the interval $[b.\mathrm{start}, b.\mathrm{end}]$, Algorithm 2 must check each subinterval defined by start and end points of obligations in B to check whether the obligatory action is authorized during that period. For this it uses the set $\mathit{subint}(B)$, which we construct as follows. Let us consider the set $\mathit{timepoints}(B)$ which is defined by $\mathit{timepoints}(\varnothing) = \varnothing$ and $\mathit{timepoints}(B \cup \{b\}) = \mathit{timepoints}(B) \cup \{b.\mathrm{start}, b.\mathrm{end}\}$. The set $\mathit{subint}(B)$ is now defined by $\mathit{subint}(B) = \{[s, e] \mid s, e \in \mathit{timepoints}(B) \wedge (s < e) \wedge \neg\exists m \in \mathit{timepoints}(B).(s < m) \wedge (m < e)\}$.

Algorithm 2 uses procedure *hasRole* (Algorithm 3) to determine whether a user's role memberships conform to requirements for $b.a$ to be authorized. Recall that to be authorized, a grant action $grant(u, r_t, u_t)$ requires that $u_t \vDash_\gamma c$, in which c is given by a *can_assign* rule, $\langle r_a, c, r_t \rangle \in \psi.CA$. This requires the ability to test that u_t is *not* in certain roles, as required to satisfy a query such as $u \vDash_\gamma \neg r$. Thus *hasRole* takes a query and the time interval of b, during which the query should be satisfied. When a pending obligation $b' \in B$ can change whether the query is satisfied during the time interval of b, $b.a$ is not guaranteed to be authorized during its full time interval, indicating that strong accountability is not satisfied. Otherwise, the current policy ($\gamma.UA$) is investigated to determine whether the query is satisfied at present. Then the last pending obligation scheduled to be performed before b that affects the role membership in question is found and inspected. The result of *hasRole* is then determined on this basis.

The procedure, *hasRole* (see page 43), must find the last grant or revoke of the given role to the given user. To support this, we use a modified interval search tree, which performs such lookups in time $\mathcal{O}(\log n)$, in which $n = |B|$. Using it, the time complexity of *hasRole* is also $\mathcal{O}(\log n)$. For each time interval in $\mathit{subint}(B \cup \{b.\mathrm{start}, b.\mathrm{end}\})$ that overlaps $[b.\mathrm{start}, b.\mathrm{end}]$, *Authorized* calls *hasRole* once for each policy rule and once for each literal in the associated constraint c, making the time complexity of *Authorized* $\mathcal{O}(qmn \log n)$, in which q is the number of policy rules

Algorithm 2 *Authorized* (γ, ψ, B, b)

Input: A policy $\langle\gamma, \psi\rangle$, an obligation set B, and an obligation b.
Output: returns **true** if b is authorized with respect to $\gamma.UA$ and B

```
1: if $b = \langle u, grant, \langle r_t, u_t\rangle, [start, end]\rangle$ then
2:     return $(\forall[s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
       $(overlaps([s,e],[\text{start},\text{end}]) \rightarrow$
         $(\exists\langle r_a, c, r_t\rangle \in \psi.CA).(hasRole(\gamma, \psi, B, u \vDash r_a, [s,e])$
         $\wedge(\forall l \in c).(hasRole(\gamma, \psi, B, u_t \vDash l, [s,e]))))$
3: else if $b = \langle u, revoke, \langle r_t, u_t\rangle, [start, end]\rangle$ then
4:     return $(\forall[s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
       $(overlaps([s,e],[\text{start},\text{end}]) \rightarrow$
         $(\exists\langle r_a, r_t\rangle \in \psi.CR).(hasRole(\gamma, \psi, B, u \vDash r_a, [s,e]))$
5: else /* $b = \langle u, a, \langle\vec{o}\rangle, [start, end]\rangle$ */
6:     return $(\forall[s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
       $(overlaps([s,e],[\text{start},\text{end}]) \rightarrow$
         $(\exists\langle r_a, \langle a, \vec{o}\rangle\rangle \in \gamma.PA).$
         $(hasRole(\gamma, \psi, B, u \vDash r_a, [s,e]))$
```

$(q = Max\{|CA|, |CR|, |PA|\})$ and m is the maximum size of the role constraints in the *can_assign* rules in CA. In the worst case, in which the obligatory action is administrative (*grant* or *revoke*), the main procedure *StrongAccountable* calls *Authorized* n times. This results in an overall worst-case time complexity of $\mathcal{O}(qmn^2 \log n)$. When the obligation to be added, b, is non-administrative, the time complexity of *StrongAccountable* is reduced to $\mathcal{O}(qmn \log n)$. The memory complexity, when implemented with a modified interval search tree is $\mathcal{O}(|\mathcal{R}|\cdot|\mathcal{U}|+n)$.

Theorem 13. *Given a set of obligations B that is known to be strongly accountable, a mini-ARBAC policy ψ, a mini-RBAC authorization state γ including an initial authorization state and a new obligation b. Deciding whether $B \cup \{b\}$ is strongly accountable can be done in polynomial time in the size of the B, γ and ψ.*

3.2.3 Non-incremental Version

Algorithm 2 can be used in a non-incremental fashion to determine whether a given obligation set B is strongly accountable. This is achieved by adding each administrative obligation to an empty modified interval search tree and then calling *Authorized* for each obligation $b \in B$ to see whether it is authorized in the context of $\langle\gamma, \psi\rangle$ and B. This can be done in $\mathcal{O}(qmn^2 \log n + a \log n) =$

Algorithm 3 *hasRole* $(\gamma, \psi, B, u \vDash_\gamma l, [s, e])$

```
Input: A policy $\langle\gamma,\psi\rangle$, an obligation set $B$, a query $u \vDash_\gamma l$ in which $l$ has either the form $r$ or $\neg r$, and a
    time interval $[s,e]$.
Output: Returns true if $u \vDash_\gamma l$ is guaranteed to hold throughout the interval $[s,e]$.
 1: if $l = r$ then /* positive role constraint */
 2:   if $(\exists\langle u', revoke, \langle r,u\rangle, [start, end]\rangle \in B).($
      $overlap([s,e],[start,end]))$ then
 3:     return false
 4:   if $\langle u,r\rangle \in \gamma.UA$ then
 5:     if $(\exists\langle u', revoke, \langle r,u\rangle, [start, end]\rangle \in B).(end < s)$ then
 6:       Select such a tuple so that end is maximized
 7:       if $(\exists\langle u'', grant, \langle r,u\rangle, [start', end']\rangle \in B).$
          $(start' > end \wedge end' < s)$ then
 8:         return true
 9:       else
10:         return false
11:     else
12:       return true
13:   else /* $\langle u,r\rangle \notin \gamma.UA$ */
14:     if $(\exists\langle u', grant, \langle r,u\rangle, [start, end]\rangle \in B).$
        $(end < s)$ then
15:       Select such a tuple so that start is maximized
16:       if $(\exists\langle u'', revoke, \langle r,u\rangle, [start', end']\rangle \in B).$
          $(overlap([start,e],[start',end']))$ then
17:         return false
18:       else
19:         return true
20:     else
21:       return false
22: else /* $l = \neg r$ negative role constraint */
23:   In case of negative role checking ($u \vDash_\gamma l$ where $l = \neg r$), the algorithm follows similar steps, reversing
      the roles of "GRANT" and "REVOKE" and reversing the negative and positive role tests.
```

$\mathcal{O}(qmn^2 \log n)$, in which a is the number of obligations to perform administrative actions ($a \leq n$). (The term $a \log n$ is the cost of constructing the search tree.)

3.2.4 Generalizing the Authorization Model (Extended mini-HRU Access Control Matrix Model)

Algorithm 1, without modifying the time complexity, can be extended to support administrative actions that modify the PA, CA and CR components of the authorization state. We believe greater generalization is also possible, for instance, to support ARBAC/RBAC models that support role hierarchies and in which the role hierarchy can be modified. However, doing so seems likely to increase the algorithm's complexity by a factor of the number of roles in the system.

In addition, algorithm 1 can be also extended to support other authorization models. In this section, we instantiate the abstract obligation model, presented in chapter 2, to use an modified version of the extended mini-HRU Access Control Matrix Model as the authorization model.

3.2.4.1 Algorithm for dealing with the extended mini-HRU Access Control Matrix Model

This section presents an incremental algorithm for deciding whether adding a new obligation to a strongly accountable obligation pool preserves the strong accountability property. Algorithm 4 is designed to be used incrementally for maintaining strong accountability. It takes as input a strongly accountable set of obligations B, an extended mini-HRU Access Control Matrix Model γ, and a new obligation b. It returns *true* if adding b to B preserves strong accountability, and *false*, otherwise.

Algorithm 4 *StrongAccountable* (γ, B, b)

```
Input: An extended mini-HRU Access Control Matrix Model $\gamma$, a strongly accountable obligation set $B$,
    and a new obligation $b$.
Output: returns true if addition of $b$ to the system preserves strong accountability.
 1: if Authorized ($\gamma$, $B$, $b$)= false then
 2:     return false
 3: if $b.a \in \gamma.\mathcal{NA}$ then
 4:     return true
 5: After:=$\{b' | b' \in B \wedge b'$.end> $b$.start$\}$
 6: $B' := B \cup b$
 7: for each obligation $b^* \in$After do
 8:     if Authorized ($\gamma$, $B'$ , $b^*$)=false then
 9:         return false
10: $B := B'$
11: return true
```

The idea behind Algorithm 4 is very similar to the Algorithm 1. It tries to determine whether adding a new obligation b to B preserves strong accountability. To this end, Algorithm 4 checks all the obligations that can be performed prior to b to decide whether b will be authorized during its entire time interval (*i.e.*, the obligation is satisfying all the preconditions in the command),

[b.start, b.end], or not. In order to check if an obligation is authorized, Algorithm 4 uses procedure *Authorized* (Algorithm 5).

Note that, if $b.a$ is an administrative command, the algorithm also needs to determine whether b may interfere with authorizations required by later obligations b^* ($b^* \in B \wedge b^*$.end > b.start). In the case of $b.a$ being a non-administrative command ($b.a \in \gamma.\mathcal{NA}$), this step is not necessary, since non-administrative commands do not alter the authorization state.

Algorithm 5 uses the same idea of Algorithm 2. But now, instead we check whether $b.u$'s rights changed during the interval $[b.\text{start}, b.\text{end}]$. To this, Algorithm 5 checks each subinterval defined by start and end points of the obligations in B to check whether the obligatory action is authorized during that period. For this, it uses the set $\mathit{subint}(B)$, which is constructed in the same way as explained for Algorithm 2

Algorithm 5 uses the procedure *hasPermission* (Algorithm 6) to determine whether a user has the necessary permissions to perform the command defined by $b.a$. To be authorized an obligation must satisfy the preconditions defined in the command rule of the action $b.a$. For this, it is necessary to check whether a user has a particular right over an object in a particular time interval. Thus, *hasPermission* takes a permission (*i.e.*, $m = (u, o, r)$) and the time interval of b, during which the permission should be satisfied. When a pending obligation $b' \in B$ can change whether the permission is satisfied during the time interval of b, $b.a$ is not guaranteed to be authorized during its full time interval, indicating that strong accountability is not satisfied. Otherwise, the current matrix of rights ($\gamma.M$) is investigated to determine whether the permission is satisfied at present. Then the last pending obligation scheduled to be performed before b that affects the cell in question is found and inspected. The result of *hasRole* is then determined on this basis. Note that, *hasRole* utilizes an auxiliary boolean function $overlap$, which receives two time interval as input and checks whether they overlap or not.

The procedure, *hasPermission*, must find the last obligation that "enters" or "removes" a given right over an object to the given user. To support this, we use a modified interval search tree (denoted by $\mathit{tree}(B)$). To build the tree we inspect each administrative obligation and add all the

Algorithm 5 *Authorized* (γ, B, b)

```
Input: An extended mini-HRU Access Control Matrix Model $\gamma$, an obligation set $B$, and an obligation $b$.
Output: returns true if $b$ is authorized with respect to $\gamma.UA$ and $B$
1: return $(\forall [s,e] \in subint(B \cup \{b.\text{start}, b.\text{end}\})).$
       $(overlaps([s,e],[b.\text{start}, b.\text{end}]) \rightarrow (\exists \alpha \in \gamma.\mathcal{AC} \cup \gamma.\mathcal{NA}).(\alpha = b.a) \wedge$
       $(\forall l \in \alpha.c).(hasPermission(\gamma, B, l, [s,e])))$
```

primitives related to $b.a$ in the tree (*i.e.*, enter or remove a right on an object to/from a user). Note that the primitives contain the action, the rights, the objects, the target users, and time windows that are related to the administrative obligations. One can perform lookups in the tree in time $\mathcal{O}(\log k)$, in which k is equal the total number of primitive operations generated by all administrative obligations in B. Recall that each administrative obligation is associated with a command that may contain more than one primitive operation.

By using the above interval search tree, the time complexity of *hasPermission* is also $\mathcal{O}(\log k)$. For each time interval in $subint(B \cup \{b.\text{start}, b.\text{end}\})$ that overlaps $[b.\text{start}, b.\text{end}]$, *Authorized* calls *hasPermission* once for each policy rule and once for each literal in the associated constraint c, making the time complexity of *Authorized* $\mathcal{O}(qmn \log k)$, in which q is the number of policy rules ($q = Max\{|\mathcal{NA}|, |\mathcal{AC}|\}$) and m is the maximum size of the preconditions in the $\mathcal{NA}$ and $\mathcal{AC}$, and $n = |B|$. In the worst case, in which the obligatory action is administrative (*grant* or *revoke*), the main procedure *StrongAccountable* calls *Authorized* n times, once for each obligation scheduled after the obligation in question. This results in an overall worst-case time complexity of $\mathcal{O}(qmn^2 \log k)$. When the obligation to be added, b, is non-administrative, the time complexity of *StrongAccountable* is reduced to $\mathcal{O}(qmn \log k)$. The space complexity of this algorithm, when implemented with a modified interval search tree, is $\mathcal{O}(|\mathcal{R}| \cdot |\mathcal{U}| \cdot |\mathcal{O}| + n)$. Note that, the above a algorithm assumes that policy conditions are purely conjunctive. This corresponds to the same assumption we make in the algorithm presented above for mini-ARBAC/mini-RBAC. However, the above algorithm also assumes that each obligatory action may add or remove one or more rights from the matrix cells. Although, Algorithm 1 and Algorithm 4 have the same complexity cost, because Algorithm 4 accept multiple modifications of rights per command; one needs to do

Algorithm 6 *hasPermission* $(\gamma, B, l, [s, e])$

Input: An extended mini-HRU access control model γ, an obligation set B, a permission l to be queried in which l has either the form $m = \langle u, o, r\rangle$ or $\neg m = \langle u, o, \neg r\rangle$, and a time interval $[s, e]$.

Output: Returns **true** iff l is guaranteed to hold throughout the interval $[s, e]$.

```
 1: if $l = m$ then /* positive permission */
 2:   if $(\exists\langle u', remove, \langle u,o,r\rangle, [start, end]\rangle \in tree(B)).($
      $overlap([s,e],[start,end]))$ then
 3:     return false
 4:   if $r \in \gamma.M[u,o]$ then
 5:     if $(\exists\langle u', remove, \langle u,o,r\rangle, [start, end]\rangle \in tree(B)).(end < s)$ then
 6:       Select such a [start, end] so that end is maximized
 7:       if $(\exists\langle u'', enter, \langle u,o,r\rangle, [start', end']\rangle \in tree(B)).$
          $(start' > end \wedge end' < s)$ then
 8:         return true
 9:       else
10:         return false
11:     else
12:       return true
13:   else /* $r$ is not originally in $\gamma.M(u,o)$ */
14:     if $(\exists\langle u', enter, \langle u,o,r\rangle, [start, end]\rangle \in tree(B)).$
        $(end < s)$ then
15:       Select such a [start, end] so that start is maximized
16:       if $(\exists\langle u'', remove, \langle u,o,r\rangle, [start', end']\rangle \in tree(B)).$
          $(overlap([start,e],[start',end']))$ then
17:         return false
18:       else
19:         return true
20:     else
21:       return false
22: else /* $l = \neg m$ */
23:   In case of negative right checking ($l = \neg m$), the algorithm follows similar steps, reversing the roles
      of "enter" and "remove" and reversing the negative and positive right tests.
```

k look ups on the data structure. Consequently, the cost of the Algorithm 4 will be greater or equal Algorithm 1.

The algorithm presented by Irwin *et al.* [31] differs from this extended mini-HRU Access Control Matrix Model variant of our algorithm by allowing disjunctions as well as conjunctions in policy-rule conditions. So, Irwin's algorithm supports an obligation model that is strictly more expressive than that supported by ours. The HRU Irwin's algorithm runs in time $\mathcal{O}(z^2n^4)$. The additional expressivity of obligations supported by the Irwin algorithm explains a factor of $\mathcal{O}(zn)$ in the difference between the complexities of these algorithms. The remaining factor of $\mathcal{O}(n/\log n)$

arises because we obtain a performance improvement by using a data structures based on binary interval search tree. Were a similar data structure used in the Irwin's algorithm, this difference would disappear.

Algorithm 4 can also be used in a non-incremental fashion to determine whether a given obligation set B is strongly accountable. This is achieved by adding each primitive operations of all the administrative obligations to an empty modified interval search tree $tree(B)$ and then calling *Authorized* for each obligation $b \in B$ to see whether it is authorized in the context of γ and B. This can be done in $\mathcal{O}(qmn^2 \log k + a \log k) = \mathcal{O}(qmn^2 \log k)$, in which a is the number of administrative obligations. (The term $a \log k$ is the cost of constructing the search tree, and k is the number of primitive operations that alter the authorization state.)

3.3 Weak Accountability

Given a schedule $b_{1..n}$, a proper prefix $b_{1..k}$ is a *critical prefix* if for all j such that $k < j \leq n$, b_k.end $\leq b_j$.end. The intuition behind the critical prefix definition is that if an obligation is attempted during the final subinterval prior to its end time, it must be authorized. However, if it is attempted before then, it need not be authorized, in which case, the reference monitor will prevent its being performed. This is acceptable because weak accountability is intended to guarantee that obligatory actions are authorized in their final subinterval. Weak accountability requires much the same thing as strong accountability, but only for critical prefixes.

Definition 14 (Weak accountability). *Given a state $s_1 \in \mathcal{S}$ and a policy $\mathcal{P}$, we say that s_1 is* weakly accountable *if for every valid schedule, $b_{1..n}$, and every critical prefix of it, $b_{1..k}$, for every policy-rule sequence $p_{1..k} \subseteq \mathcal{P}$ and state s_{k+1} such that $s_1 \xrightarrow{\langle b,p \rangle_{1..k}} s_{k+1}$, there exists a policy rule p_{k+1} and a state s_{k+2} such that $s_{k+1} \xrightarrow{\langle b,p \rangle_{k+1}} s_{k+2}$.*

Example 15 (Weak Accountability). *Using the mini-RBAC, mini-ARBAC policies presented in chapter 2 tables 2.1 and 2.2, let us assume we have two pending obligations b_1 and b_2 defined as follows:*

- b_1 : *Joan must grant developer role to Carl in time* $[7, 9]$
- b_2 : *Carl must develop sourceCode in time* $[5, 20]$

The example obligation system is weakly accountable, but not strongly accountable. It is not strongly accountable as obligation b_2 is not authorized before time 7 and is not guaranteed to be authorized until time 9. Initially Carl does not have the role developer (table 2.1) and Joan may not grant him the role until time 9. On the other hand, the system is weakly accountable because obligation b_1 can be fulfilled any time in its time interval, since Joan already has the role of security manager and Carl satisfies the CA rule constraints, and b_2 can be fulfilled anytime after time 9.

3.3.1 The Weak Accountability Problem

An *instance of the weak accountability problem* is given by a tuple $\langle \gamma, \psi, B \rangle$ in which:

- $\gamma = \langle R, \mathit{UA}, \mathit{PA} \rangle$ is an initial mini-RBAC authorization state.
- $\psi = \langle \mathit{CA}, \mathit{CR} \rangle$ is a mini-ARBAC policy.
- B is a set of pending obligations.

Theorem 16. *Given a set of obligations B, a mini-ARBAC policy ψ, a mini-RBAC authorization state γ, and an initial authorization state γ, deciding whether the given system is weakly accountable is co-NP complete in the size of the B, γ and ψ.*

Proof. We are given a 3-SAT expression $S = d_1 \wedge d_2 \wedge \cdots \wedge d_n$ where each $d_i = l_1^i \vee l_2^i \vee l_3^i$ for $i \in [1,\ n]$. Let the set of variables involved in this expression be given by $X = \{x_1, x_2, \ldots, x_m\}$. Thus each literal $l_j^i = x_k$ or $\neg x_k$ where $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$.

We construct a system of obligations within the context of mini-ARBAC. This system will be weakly accountable if and only if S is not satisfiable. The system has two administrative roles, ar_1 and ar_2. For every disjunct d_i of S we have a corresponding role r_{d_i} and for every variable x_k of S we have an associated role r_{x_k} in the system. Along with these roles, the system has two more roles which are important to us r' and r_{goal}.

We assume that the system has three users, u_0, u_1 and u_2. The initial user-role assignment in the system is as follows:

$$\begin{gathered}
\textit{user\_assignment}(u_0, \emptyset) \\
\textit{user\_assignment}(u_1, \{ar_1\}) \\
\textit{user\_assignment}(u_2, \{ar_2\})
\end{gathered}$$

We ignore the role-permission assignment rules, as it has no impact on the argument. Thus the policy rules are as follows:

$$\begin{gathered}
\textit{can\_assign}(ar_1, \textit{true}, r_{x_1}) \\
\vdots \\
\textit{can\_assign}(ar_1, \textit{true}, r_{x_m}) \\
\textit{can\_revoke}(ar_1, r_{x_1}) \\
\vdots \\
\textit{can\_revoke}(ar_1, r_{x_m})
\end{gathered}$$

We also add one *can_assign* rule for each l_j^i, the form of which depends on the form of l_j^i as follows:

$$\textit{can\_assign}(ar_1,\ r_{x_k},\ r_{d_i}) \qquad when\ l_j^i\ =\ x_k \tag{3.1}$$

$$for\ some\ k \in [1,\ m],\ j \in [1,\ 3]\ and\ i \in [1,\ n].$$

$$\textit{can\_assign}(ar_1,\ \neg r_{x_k},\ r_{d_i}) \qquad when\ l_j^i\ = \neg x_k \tag{3.2}$$

$$for\ some\ k \in [1,\ m],\ j \in [1,\ 3]\ and\ i \in [1,\ n].$$

$$\textit{can\_assign}(ar_1, \bigwedge_{i=1}^{n} r_{d_i}, r') \tag{3.3}$$

$$\textit{can\_assign}(ar_1, \neg r', r_{goal}) \tag{3.4}$$

The system currently has some pending obligations where each obligation is of form

$$\begin{array}{c} \textit{can\_assign}(ar_2,\, \textit{true},\, r_{d_1}) \\ \vdots \\ \textit{can\_assign}(ar_2,\, \textit{true},\, r_{d_n}) \\ \textit{can\_assign}(ar_2,\, \textit{true},\, r_{x_1}) \\ \vdots \\ \textit{can\_assign}(ar_2,\, \textit{true},\, r_{x_m}) \\ \textit{can\_revoke}(ar_2,\, r_{x_1}) \\ \vdots \\ \textit{can\_revoke}(ar_2,\, r_{x_m}) \end{array}$$

$(action,\ [t_s,\ t_e])$. Intuitively, the first ones nondeterministically select a truth assignment.

$$\begin{array}{rcl} b_{1,\,t} & = & (u_1\ Grants\ r_{x_1}\ to\ u_0,\ [1,\ 2]) \\ b_{1,\,f} & = & (u_1\ Revokes\ r_{x_1}\ from\ u_0,\ [1,\ 2]) \\ & \vdots & \\ b_{m,\,t} & = & (u_1\ Grants\ r_{x_m}\ to\ u_0,\ [1,\ 2]) \\ b_{m,\,f} & = & (u_1\ Revokes\ r_{x_m}\ from\ u_0,\ [1,\ 2]) \end{array}$$

For each literal l^i_j we add one obligation. The time interval of this obligation depends on whether the literal is positive or negative, which assists in ensuring the obligations can be fulfilled during the cleanup phase.

$$b_{l^i_j} = \begin{cases} (u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 21]) & \text{if } l^i_j = x_k \\ (u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 31]) & \text{if } l^i_j = \neg x_k \end{cases}$$

The time line during which all the obligations in the system should be fulfilled can be divided into 6 time periods as shown in Table 3.1.

The logic behind the reduction is as follows. The obligations $b_{k,\,t}$ and $b_{k,\,f}$ for $k \in [1,\ m]$ can always be executed before their deadline. Obligations $b^{cleanup}_{d_i}$, $b^{cleanup}_{x_k}$ and $b^{cleanup}_{\neg x_k}$ for $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$ can also be executed before their stipulated deadline enabling b^* and each $b_{l^i_j}$ to be carried out eventually. Now whether b can be carried out depends on how it is scheduled with respect to obligation b^*. If b^* is performed before b then b can not be executed because b^*

$$
\begin{aligned}
b^* &= (u_1\ Grants\ r'\ to\ u_0,\ [5,\ 31]) \\
b &= (u_1\ Grants\ r_{goal}\ to\ u_0,\ [10,\ 15]) \\
b^{cleanup}_{d_1} &= (u_2\ Grants\ r_{d_1}\ to\ u_0,\ [16,\ 20]) \\
&\vdots \\
b^{cleanup}_{d_n} &= (u_2\ Grants\ r_{d_n}\ to\ u_0,\ [16,\ 20]) \\
b^{cleanup}_{x_1} &= (u_2\ Grants\ r_{x_1}\ to\ u_0,\ [16,\ 20]) \\
&\vdots \\
b^{cleanup}_{x_m} &= (u_2\ Grants\ r_{x_m}\ to\ u_0,\ [16,\ 20]) \\
b^{cleanup}_{\neg x_1} &= (u_2\ Revokes\ r_{x_1}\ from\ u_0,\ [22,\ 30]) \\
&\vdots \\
b^{cleanup}_{\neg x_m} &= (u_2\ Revokes\ r_{x_m}\ from\ u_0,\ [22,\ 30])
\end{aligned}
$$

adds u_0 to a role that prevents u_0 to receive the role granted in b. We show that S is satisfiable if and only if there exist a schedule in which b^* is performed before b; resulting the system being in weakly unaccountable state.

For the *only if* part, suppose the 3-SAT problem is satisfiable. We construct a schedule that demonstrates the system is not weakly accountable. Such a schedule begins in period 1 by selecting an assignment that satisfies the 3-SAT problem, representing this in the assignment of each r_{x_k} to u_0 just in case x_k is true. Then, in period 2, u_1 grants each r_{d_i} to u_0, as permitted by statements (3.1) and (3.2). In period 3, u_1 grants r' to u_0, as permitted by statement (3.3). Finally, in period 4, u_1 is unable to fulfill obligation b due to statement (3.4)

For the *if* part of our reduction, we show the converse. Suppose there is no satisfying assignment for S. In this case, there is no scheduling of obligations $b_{1,t} \cdots b_{m,t}$ and $b_{1,f} \cdots b_{m,f}$ such that every $b_{l^i_j}$ can be performed before time 14. Thus, b^* can not be performed before time 15. This ensures b can execute in every schedule. The pending obligations at time 15 are then each $b_{l^i_j}$ and b^*. By time 20, u_2 must grant each r_{d_i} to u_0, enabling u_1 to fulfill b^*. Also by time 20, u_2 must grant each r_{x_k} to u_0, enabling u_1 to fulfill each obligation of form $(u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 21])$. Then, by time 30, u_2 must revoke each r_{x_k} from u_0 enabling u_1 to fulfill each obligation of form $(u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 31])$. Thus all the obligations can be carried out, resulting the system being in a weakly accountable state. We can say that, when there is no satisfying assignment for S, the system is in weakly accountable state.

Table 3.1: Time periods in the proof of Theorem 16.

Period	Interval	Activity
1	1...2	Nondeterministically select a truth assignment. The represented assignments make x_k true just in case u_0 has role r_{x_k}.
2	3...4	It is possible to grant r_{d_i} to u_0 during this period just in case one of the literals in d_i is true under the selected assignment.
3	5...9	It is possible that u_0 gets r' during this period (via obligation b^*) just in case each disjunction in the 3-SAT problem is satisfied.
4	10...15	u_1 is unable to fulfill obligation b just in case u has been assigned r' in period 3 (via obligation b^*).
5	16...21	In this period, u_2 grants each of the r_{d_i}'s to u_0 (via obligation $b_{d_i}^{cleanup}$) enabling obligation b^* to be fulfilled at its last time period if b^* has not been carried out before. Moreover u_2 also grants each r_{x_k} to u_0 (via obligation $b_{x_k}^{cleanup}$) enabling the obligation $b_{l_j^i}$ where $l_j^i = x_k$ to be fulfilled in time 21 if it has not been fulfilled before.
6	22...31	Here u_2 revokes each r_{x_k} from u_0 (via obligation $b_{\neg x_k}^{cleanup}$) enabling the obligation $b_{l_j^i}$ where $l_j^i = \neg x_k$ to be fulfilled in time 31 if it has not been fulfilled before.

The above reduction shows that checking weak accountability in a system of obligations governed by mini-ARBAC policy is *co-NP Hard*. Now we want to show that checking weak accountability in such a system is in *co-NP*. A nondeterministic algorithm to identify systems that are not weakly accountable begins by guessing a schedule of the obligations indicating for each obligation when it is to be carried out, then we can check if all the obligations of the system are present in the schedule. Then for each obligation, we check if it is authorized. If it is authorized, we simulate the behavior of the obligation and update the authorization state of the system. Otherwise we check if the obligation is in its last time. If it is, we stop and declare the system is not weakly accountable. So given a schedule of the obligations, we can check if the schedule is a valid one in $\mathcal{O}(n)$. Thus checking weak accountability in a system of obligations governed by mini-ARBAC is *co-NP Complete*. □

The approaches we use for solving weak accountability are, respectively, an algorithm designed specifically for this purpose and the general-purpose technique of model checking. Model checking is a formal verification method for determining whether a FSM model satisfies a temporal logic property.

The special-purpose algorithm explicitly considers all authorized[2] critical prefixes of valid schedules and checks whether the next obligation in the schedule is authorized. On the other hand, the model checking approach models the execution of a set of obligations as a finite state machine (FSM) and checks the accountability property, as specified by a temporal logic formula. We use a symbolic [16] model checker (*viz.*, Cadence SMV [6]). This approach has the advantage that, without constructing actual traces, it computes a characterization of states starting from which a trace can reach a state that violates accountability. An efficient dynamic programming algorithm avoids considering multiple interleavings of actions when order makes no difference to the result.

When the set is not weakly accountable, both methods generate a counter example: if a given state s_0 is not weakly accountable, then a *counter example* is an authorized critical prefix $b_{0..k}$ of a valid schedule such that the next obligation in the schedule is not authorized.

3.3.2 Special-Purpose Algorithm for Determining Weak Accountability

In this section, we present an algorithm (see algorithm 7) designed specifically to solve the weak accountability problem(see section 3.3.1). The algorithm takes as input a list of pending obligations $B = b_0, \ldots, b_{n-1}$, $n = |B|$, a mini-RBAC authorization state γ, and a mini-ARBAC policy ψ. It returns *true* if the list of pending obligations B is weakly accountable and *false* otherwise.

As mentioned above, the algorithm (see algorithm 7) investigates all authorized critical prefixes of valid schedules of B and checks whether the next obligation in the schedule is authorized. If not, a counter example has been found and is returned. Otherwise, it returns *true*, indicating that B is weakly accountable.

[2]Given a state s_0 and a valid schedule of $s_0.B$, a critical prefix $b_{0..k}$ of that schedule is *authorized* if there exists a policy-rule sequence $p_{0..k} \subseteq \mathcal{P}$ and state s_{k+1} satisfying $s_0 \overset{\langle b,p \rangle_{0..k}}{\longrightarrow} s_{k+1}$.

Algorithm 7 *WeaklyAccountable*(γ, ψ, B)

```
Input: A policy $\langle \gamma, \psi \rangle$, current pending obligations $B$
Output: return true $B$ is weakly accountable.
1: sort the obligation list $B$ according to the non-decreasing order of end time of the obligations.
2: numEx := 0
3: executed [1...n] := false
4: T := null
5: return solve($0, \gamma, \psi, B, T, numEx, executed$)
```

The algorithm uses a recursive procedure, *solve* (see algorithm 8), to incrementally explore valid schedules in a depth-first manner. Each invocation of *solve* extends a prefix that is known to be authorized and extensible to at least one valid schedule. For each obligation that, under the validity constraint, is a candidate to extend the prefix, the algorithm determines whether that obligation is authorized in the current authorization state. (The current authorization state is maintained so as to reflect the initial authorization state provided by γ and the effect on it of obligations already in the prefix.) If the obligation is authorized, it is appended to the prefix, the authorization state is updated, and the procedure is invoked recursively to explore further extensions of the prefix. If it is not authorized, the algorithm determines whether the obligation's end time is later than that of some other obligation that is not in the current prefix. If so, the obligation is skipped, as the current prefix is not a critical prefix of schedules in which this obligation comes next. If not, a counter example has been found; it is reported, and the algorithm terminates. Unless a counter example has been found, the algorithm proceeds by examining any remaining candidates to be added to the current schedule.

The algorithm uses a boolean array *executed*$[1..n]$ to represent the set of obligations that have been successfully incorporated into the current partial schedule.

Recall that a valid schedule must execute b_1 before b_2 if $b_1.\mathrm{end} \leq b_2.\mathrm{start}$. As *solve* incrementally constructs a possible schedule, an obligation is not yet ready to be incorporated if some other unscheduled obligation has to go before it. On the other hand, an obligation b is ready to be scheduled if $b.\mathrm{start} \leq$ *minTime* in which *minTime*, the least end time of all unincorporated obligations, is given by *minTime* = *min*$\{b_i.\mathrm{end} \mid$ *executed*$[i]$ = *false*$\}$.

Algorithm 8 *solve*$(ready, \gamma, \psi, B, T, numEx, executed)$

Input: The index *ready*, represents the index of the last obligation that is ready, a policy $\langle \gamma \psi \rangle$, obligation array B, multi-set of end times T, total number of obligation executed so far *numEx*, status of obligations *executed*.

Output: returns **false** if there is a counter example which ensures that B is not weakly accountable.

```
1:  if $numEx \geq |B|$ then
2:      return true
3:  $i := ready + 1$
4:  if $T = null$ then
5:      $T.insert(B[i].\text{end})$
6:      $ready := ready + 1$; $i := i + 1$
7:  while $i \leq |B|$ and $T.minValue() \geq B[i].\text{start}$ do
8:      $T.insert(B[i].\text{end})$
9:      $ready := ready + 1$; $i := i + 1$
10: for all $j \in [1, ready]$ do
11:     /* For $B[j] = \langle u, a, \vec{o}, \langle s, e \rangle \rangle$, $a(u, \vec{o})(\gamma)$ is the policy state obtained after the action */
12:     if $executed[j] =$ false then
13:         if $(\exists p \in \mathcal{P}).(\gamma \vDash p.cond(u, \vec{o}, a))$ then
14:             /* obligation $B[j]$ is currently authorized */
15:             $executed[j] :=$ true
16:             $T.delete(B[j].\text{end})$
17:             if $\neg solve(ready, a(u, \vec{o})(\gamma),$
                $B, T, numEx + 1, executed)$ then
18:                 return false
19:             $executed[j] :=$ false
20:             $T.insert(B[j].\text{end})$
21:         else
22:             if $T.minValue() \geq B[j].\text{end}$ then
23:                 print counter example
24:                 return false
25: return true
```

3.3.2.1 Optimizing the Algorithm

The algorithm assumes that the obligation set is represented by an array B and begins by sorting B by time interval start time. This supports identifying obligations that are ready to be scheduled very efficiently by a single index, *ready*, which is maintained so as to preserve the invariant that *ready* is the greatest i such that $executed[i] = false \wedge b_i.\text{start} \leq minTime$.

The multi-set of end time points of as yet unscheduled obligations is implemented by using a balanced binary search tree T, which supports finding the minimum value, inserting a new value, and deleting a value, each in $\mathcal{O}(\log n)$. We represent $\gamma.UA$ by a 2-dimensional array representing its characteristic function, which enables us to perform lookup, update and restore operations in constant time.

The special-purpose weak accountability algorithm becomes impractical when the number of overlapping obligations is greater than about 10. However, in practice, usually the overlapping obligations are not related directly or indirectly. Using that intuition, we can optimize the above algorithm by using a notion of *dependence of obligations*. We define a direct dependence relations, then take its reflexive, symmetric, transitive closure to obtain an equivalence relation. This in turn enables us to partition the set of pending obligations so that if we select any pair of obligations, one from each of two distinct sets in the partition, neither obligation depends on the other. We then execute Algorithm 7 separately on each set in the partition. This often significantly reduces the size of the problem instances that must be solved. We say that one obligation is *directly dependent* on another if the first grants or revokes a role that the other obligation uses (or might use according to the policy) (*i.e.*, as a pre-condition or as part of a permission).

Example 17 (Dependency between obligations)**.** *Using the mini-RBAC, mini-ARBAC policies presented in chapter 2 tables 2.1 and 2.2, two obligations b_1 and b_2 defined as follows:*

- b_1 : *Joan must grant developer role to Carl in time* $[5, 9]$
- b_2 : *Carl must develop sourceCode in time* $[10, 20]$

b_2 depends directly on b_1, because Carl needs the developer role to develop sourceCode (b_2), and he is going to receive this from Joan (b_1).

According to the definition of dependence, we write $b_1 \sim b_2$ if there is a direct dependence between obligation b_1 and obligation b_2. We write $\approx$ for the reflexive, symmetric, transitive closure. Thus, $\approx$ is an equivalence relation over elements of B and, as such, induces a partition $\{B_1, \ldots, B_k\}$ of B.

Theorem 18. *Let $\{B_1, \ldots, B_k\}$ be the partition a pending set of obligations, B, induced by $\approx$. There is a counter example of weak accountability for one of the obligation sets B_i for some i, $1 \leq i \leq k$, if and only if there is a counter example for B.*

Proof. $\Rightarrow$ direction :

Without loss of generality, we will show that if there is a counter example of weak accountability within obligations of set B_1 then we can find a counter example of weak accountability in obligation set B.

Thus we can fix, σ^{B_1} to be a counter example of weak accountability for obligation set B_1 and let b^* refer to the obligation in the schedule σ^{B_1} that is not authorized. On the other hand, for obligation set B_2 we chose any weakly authorized schedule σ^{B_2}. We next construct an interleaving σ^B of σ^{B_1} and σ^{B_2}, and show that it is valid, weakly authorized, and b^* is not authorized in σ^B.

We start with σ^{B_2} and, starting with $\sigma_1^{B_1}$, add each element $\sigma_k^{B_1}$ to the schedule being constructed in the least position that satisfies the following properties :

1. If $k > 1$, it is after $\sigma_{k-1}^{B_1}$
2. It is placed after each $\sigma_o^{B_2}$, if any such o exists, such that $\sigma_o^{B_2}.\mathrm{end} \leq \sigma_k^{B_1}.\mathrm{start}$
3. If $\sigma_k^{B_1}$ is not authorized in σ^{B_1}, it is placed after each $\sigma_m^{B_2}$ if any such m exists, such that $\sigma_m^{B_2}.\mathrm{end} < \sigma_k^{B_1}.\mathrm{end}$

We now use induction on k to show that the schedule of $B_2 \cup \{\sigma_i^{B_1} \mid 1 \leq i \leq k\}$ constructed at stage k is valid and weakly authorized. The base case follows immediately from assumptions on σ^{B_2}. For the step, recall that the requirement of validity of a schedule ensures that each unordered pair of obligations is ordered in a manner that respects their time intervals. The induction assumption that the schedule of $(B_2 \cup \{\sigma_i^{B_1} \mid 1 \leq\ i \leq k-1\})$ that is constructed at stage $k-1$ is valid. Thus, when $\sigma_k^{B_1}$ does not occur in the pair the validity constraint is satisfied.

Let us consider the pairs that include $\sigma_k^{B_1}$. For validity, we must show that each obligation b' scheduled after $\sigma_k^{B_1}$ satisfies $\sigma_k^{B_1}.\mathrm{start} < b'.\mathrm{end}$ and each obligation, b, scheduled before $\sigma_k^{B_1}$ satisfies $b.\mathrm{start} < \sigma_k^{B_1}.\mathrm{end}$. For the former, requirement (2) ensures validity for all such pairs of obligations b' and $\sigma_k^{B_1}$. For the latter, when $b \in B_1$, this follows from requirement (1) and the assumption that σ^{B_1} is valid. For the case in which $b \in B_2$, consider the last $b' \in B_2$ in the constructed schedule that precedes $\sigma_k^{B_1}$. The $b'' \in B_1$ that immediately follows b' must satisfy one of two cases : either (a) $b'.\mathrm{end} \le b''.\mathrm{start}$ or (b) $b'.\mathrm{end} < b''.\mathrm{end}$ and b'' is not authorized in σ^{B_1}. From validity of σ^{B_2} (or from well-formedness, if $b = b'$), we have $b.\mathrm{start} < b'.\mathrm{end}$. In case (a), we use the fact that $b''.\mathrm{start} < \sigma_k^{B_1}.\mathrm{end}$ follows from the validity of σ^{B_1}, or the well-formedness when $b'' = \sigma_k^{B_1}$. In case (b), we use the fact that σ^{B_1} is weakly authorized and b'' is unauthorized in σ^{B_1} to show that $b''.\mathrm{end} < \sigma_k^{B_1}.\mathrm{end}$.

To show the schedule is weakly authorized, we must show that

1. If $\sigma_k^{B_1}$ is not authorized in σ^{B_1} then it comes after any of $\sigma_m^{B_2}$ such that $\sigma_m^{B_2}.\mathrm{end} < \sigma_k^{B_1}.\mathrm{end}$

2. If some $b \in B_2$ is not authorized in σ^{B_2} and b comes before $\sigma_k^{B_1}$, then $b.\mathrm{end} \le \sigma_k^{B_1}.\mathrm{end}$

For (1), we can see that requirement (3) achieves exactly what is required by it.

For (2), we again consider b' and b'' as in the previous part of the proof and the two cases, (a) and (b). In both cases, because b is unauthorized in σ^{B_2}, yet σ^{B_2} is weakly authorized, we have

$$b.\mathrm{end} \le b'.\mathrm{end} \tag{3.5}$$

case a: $b'.\mathrm{end} \le b''.\mathrm{start}$. Here we use the fact that σ^{B_1} is valid to obtain $b''.\mathrm{start} < \sigma_k^{B_1}.\mathrm{end}$, which yields the desired result, $b.\mathrm{end} \le \sigma_k^{B_1}.\mathrm{end}$. case b: $b'.\mathrm{end} < b''.\mathrm{end}$ and b'' is not authorized in σ^{B_1}. Now because σ^{B_1} is weakly authorized, we obtain $b''.\mathrm{end} \le \sigma_k^{B_1}.\mathrm{end}$ completing this case as well.

We now show that b^* is not authorized in σ^B. Suppose for contradiction that it were authorized. In this case, we show that there must be some obligation $b^\diamond \in \sigma^{B_2}$ that influenced the authorization state of obligation b^* to make it become authorized during the execution of obligation set B. If $b^\diamond \in \sigma^{B_1}$ then during the execution of σ^B, $b^\diamond$ will not be authorized. As $b^\diamond \in \sigma^{B_2}$, $b^\diamond$ is not authorized during the execution of σ^{B_1} whereas it is authorized during execution of σ^B.

There are several scenarios which can arise when $b^\diamond$ can influence the authorization state of b^*. Let us investigate one of the possible scenarios, when b^* is not authorized during the execution of obligation set B_1 according to the initial authorization state. There is no previous obligation in B_1 which grants the necessary permissions to obligation b^* resulting it to be not authorized. During the execution of obligation set B, $b^\diamond \in B_2$ can grant $b^* \in B_1$ the necessary permissions resulting b^* to become authorized during the execution of B.

There is another possible scenario that arise during execution of obligation B by which $b^\diamond$ can influence the authorization state of b^*. In the scenario, b^* is authorized according to the initial authorization state during the execution of B_1 but there is a previous obligation $b_r \in B_1$ which revoked the necessary permissions from b^* resulting it to be unauthorized. During the execution of B, $b^\diamond$ can grant the permission revoked by b_r after b_r resulting b^* to be authorized or $b^\diamond$ can grant some other permission which can enable b^* to be authorized using a different policy rule.

According to our definition of dependence, in all the scenarios described above the obligations $b^\diamond$ and b^* are dependent ($b^\diamond \sim b^*$) so $B_1 \approx B_2$ which is a contradiction to our assumption of $B_1 \not\approx B_2$.

Thus, when there is a counter example of weak accountability within obligations of set B_1 then there is a counter example of weak accountability present in B.

$\Leftarrow$ direction:

If there is a counter example of weak accountability within obligations of set B then there is a counter example present in either set B_1 or B_2 or both.

We assume that obligation set B is not weakly accountable and σ^B be a counter example for B. Let b^* be the obligation which is not authorized according to σ^B. For $i \in \{1, 2\}$, let σ^{B_i} enumerate B_i in the same order as it is enumerated by σ^B. Without loss of generality, we also assume $b^* \in B_1$. We show that σ^{B_1} is a counter example. If it is not then there exists an obligation $b^\diamond \in B_2$ which influenced the authorization state of obligation b^* during execution in obligation set B. Several scenarios can arise where b^*'s authorization state can be influenced by $b^\diamond$. One possible scenario is when b^* is authorized according to the initial authorization state but $b^\diamond$ scheduled before b^* revokes the necessary permissions from b^* resulting it to be unauthorized during execution of B but as $b^\diamond \notin B_1$, b^* is authorized during execution of B_1.

Another possible scenario arises when b^* is authorized according to the initial authorization state. But there is another obligation $b^{\square} \in B_1$ scheduled before b^* and $b^{\diamond} \in B_2$, that grants the necessary permissions making b^* to be authorized. But $b^{\diamond}$ scheduled before b^* and after $b^{\square}$, revokes the necessary permission from b^* resulting it to be unauthorized eventually during execution of B. As $b^{\diamond} \notin B_1$, b^* becomes authorized during execution of B_1.

Thus according to our definition of dependence between obligations, in all the scenarios mentioned here $b^{\diamond}$ and b^* can not be independent ($b^{\diamond} \sim b^*$) resulting obligation B_1 and B_2 to be dependent ($B_1 \approx B_2$), which is a contradiction to our assumption of $B_1 \not\approx B_2$.

Thus when there is a counter example of weak accountability within obligations of set B then there is a counter example present in either set B_1 or B_2 or both.

□

Even with this optimization, the special-purpose algorithm becomes impractical when it is applied to a set of obligations that have a high degree of overlapping among obligations that are also dependent on one another. In that case, the optimization gains little advantage. Fortunately, this circumstance seems to be quite rare in practice (see section 3.4).

3.3.3 Model Checking Approach for Determining Weak Accountability

In this section, we describe an approach to determining weak accountability by using model checking techniques. Model checking [18] is a formal verification method for determining whether a finite state machine (FSM) model satisfies a temporal logic property. In our study, we use the symbolic model checker, Cadence SMV [6, 16]. For each input problem instance, we construct FSMs that encode the obligations and explores all possible valid, weakly authorized schedules. In effect, it checks each one to determine whether they contain any unauthorized obligations. dAs we report below in section 3.4, this enables the technique to solve many problem instances that involve too many possible schedules for our special-purpose algorithm to handle.

3.3.3.1 Translation and Optimizations

We use a C++ program to translate a mini-ARBAC security policy ψ, a mini-RBAC authorization state γ, the initial state of a given system $\gamma.UA$, and a set of obligations B into a Cadence SMV finite-state-machine (FSM). The translator hard-codes the security policy $\langle \gamma, \psi \rangle$ and the obligation set into a FSM. Then, we execute the SMV code to determine whether the obligation set is weak accountable (Figure 3.1).

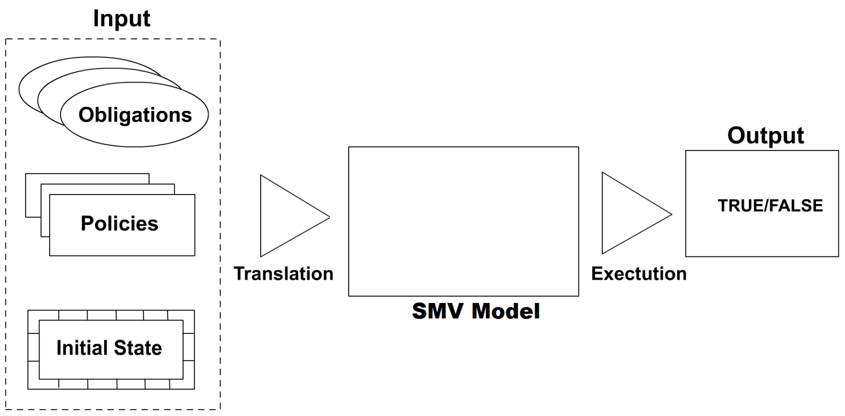

Figure 3.1: Translation model

To reduce the number of states generated, the translator scans the obligations and represents as a constant each entry of the authorization state that is not modified during execution. For optimization purposes, instead of the system time assuming arbitrary values; we normalize the end point of each obligation to differ by one. To decrease the number of the states generated, we also make sure that the nonadministrative obligations are scheduled just in their end times as they do not alter the authorization state of the system.

3.3.3.2 FSMs State Variables

The FSMs we construct contain the following 5 state variables, which are finite.

• t (Integer) — t represents the current system time. Its value ascends through the start and end points of input-obligation time intervals. It is used to identify obligations that are ready to execute (their time interval has been entered) and those that must be executed before time advances (their time interval is about to end).

• n (Integer) — This represents the index of the obligation that is selected by the scheduler to be executed.

• obl (Array of boolean) — Indicates for each obligation i whether it has been performed (0 for no, 1 for yes).

• UA (Two dimensional array of boolean) — UA represents the current authorization state(*i.e.*, the same UA as depicted in the mini-RBAC model). $UA[u][r]$ takes the value 1 when user u is in role r, and 0 otherwise.

• aco (Boolean) — Indicates the system is weakly accountable.

3.3.3.3 The FSM for Checking Weak Accountability

The FSM is divided in two main modules as pictured in Figure 3.2:

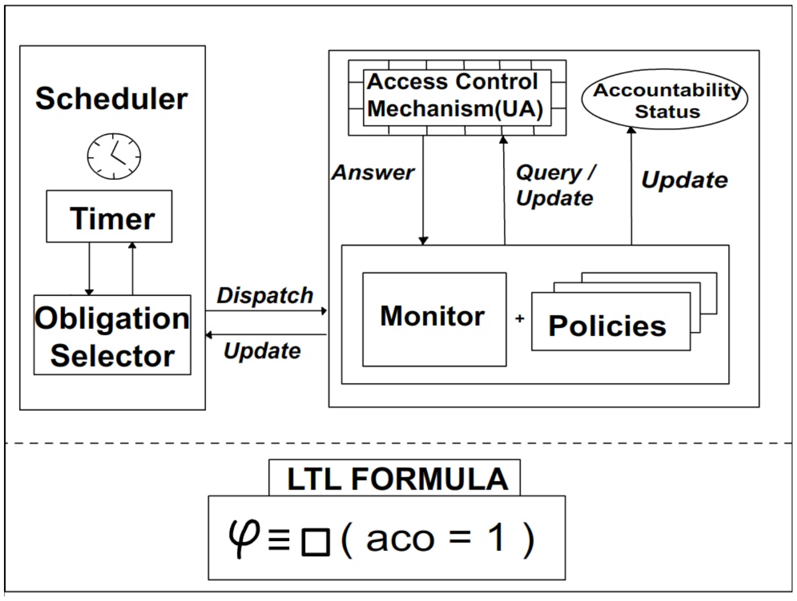

Figure 3.2: SMV model for an obligation system

The scheduler comprises two parts. *The timer* increments the system time when the end times of the remaining obligations all exceed the current system time. *The obligation selector* nondeterministically selects one of the pending obligations that is currently ready to be executed. An obligation b is ready at time t if it satisfies $t \geq b.\text{start} \wedge t \leq b.\text{end}$.

The monitor receives an obligation b from the scheduler, and checks whether it is authorized according to UA. If so, it records that this obligation has been fulfilled by setting $obl[b]$ =1. If b is an administrative action, the monitor changes UA accordingly. On the other hand, if b is not authorized, this may or may not represent a counterexample to weak accountability. If $t < b$.end, weak accountability can still be satisfied if b becomes authorized later, before the end of its time interval. In this case, the monitor returns control to the scheduler to select another obligation. However, if $t = b$.end, the monitor sets $aco = 0$, signaling that the system is not weak accountable that is captured by checking whether it is always the case that the variable aco is true.

3.3.4 Weak Accountability under the original HRU Access Control Matrix Model

In this section we present a proof that the accountability problem remains co-NP complete when the authorization system is restricted to use only the expressive power of the original HRU access matrix model. Recall that in the original HRU access matrix model commands have unique names, and commands' preconditions are only given by a conjunction of positive permissions. Note that, the proof presented here strengths the theoretical results presented in [31], which had used super set of the original HRU Access Control Matrix Model.

Theorem 19. *Given a set of obligations B, a HRU Access Control Matrix Model γ, deciding whether the given system is weakly accountable is co-NP complete in the size of the B and γ.*

Proof. We are given a 3-SAT expression $S = d_1 \wedge d_2 \wedge \cdots \wedge d_n$ where each $d_i = l_1^i \vee l_2^i \vee l_3^i$ for $i \in [1,\ n]$. Let the set of variables involved in this expression be given by $X = \{x_1, x_2, \ldots, x_m\}$. Thus each literal $l_j^i = x_k$ or $\neg x_k$ where $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$.

We construct a system of obligations within the context of HRU Access Control Matrix Model. This system will be weakly accountable if and only if S is not satisfiable. The system has just one object o, and all the rights are related to this object. For every disjunct d_i of S we have a corresponding right r_{d_i} and for every variable x_k of S we have an associated right r_{x_k}. We represent the lack of the right r_{x_k} as another right called Nr_{x_k}. Along with these rights, the system has five more rights which are important to us r_{goal}, ar_1, ar_2, r'. Again we represent the lack of the right r' as another right called Nr'.

We assume that the system has three users, u_0, u_1 and u_2. The initial matrix of rights to object o in the system is as follows:

$$\begin{gathered} M[u_0, o] = \{Nr', Nr_{x_1}, \ldots, Nr_{x_m}\} \\ M[u_1, o] = \{ar_1, Nr'\} \\ M[u_2, o] = \{ar_2, Nr'\} \end{gathered}$$

In our policy rules we have one *grant* command for each l_j^i, the form of which depends on the form of l_j^i as follows:

```
command $grant \bullet r_{d_i}(u,t,o)$        $when\ l_j^i\ =\ x_k$                    (3.6)
  if $M[u,o] = ar_1\ \wedge\ M[t,o] = r_{x_k}$ then
      enter $r_{d_i}$ into $M[t,o]$
    $for\ some\ k \in [1,\ m],\ j \in [1,\ 3]\ and\ i \in [1,\ n].$
```

```
command $grant \bullet r_{d_i}(u,t,o)$        $when\ l_j^i\ =\ \neg x_k$                 (3.7)
  if $M[u,o] = ar_1\ \wedge\ M[t,o] = Nr_{x_k}$ then
    enter $r_{d_i}$ into $M[t,o]$
      $for\ some\ k \in [1,\ m],\ j \in [1,\ 3]\ and\ i \in [1,\ n].$
```

```
command $grant \bullet r'(u,t,o)$                                                     (3.8)
  if $M[u,o]\ =\ \bigwedge_{i=1}^{n} r_{d_i}$ then
    enter $r'$ into $M[t,o]$
```

$$\begin{aligned} &\textbf{command } \mathit{grant} \bullet r_{goal}(u,t,o) \\ &\quad \textbf{if } M[u,o] \;=\; Nr' \textbf{ then} \\ &\qquad \textbf{enter } r_{goal} \textbf{ into } M[t,o] \end{aligned} \tag{3.9}$$

We also have these commands in the policy rules:

command $\mathit{grant} \bullet r_{x_1}(u,t,o)$ **if** $M[u,o] = ar_1$ **then** **enter** r_{x_1} **into** $M[t,o]$ **remove** Nr_{x_1} **from** $M[t,o]$	**command** $\mathit{grant} \bullet r_{x_1} \bullet \mathit{2}(u,t,o)$ **if** $M[u,o] = ar_2$ **then** **enter** r_{x_1} **into** $M[t,o]$ **remove** Nr_{x_1} **from** $M[t,o]$
$\vdots$	$\vdots$
command $\mathit{grant} \bullet r_{x_m}(u,t,o)$ **if** $M[u,o] = ar_1$ **then** **enter** r_{x_m} **into** $M[t,o]$ **remove** Nr_{x_m} **from** $M[t,o]$	**command** $\mathit{grant} \bullet r_{x_m} \bullet \mathit{2}(u,t,o)$ **if** $M[u,o] = ar_2$ **then** **enter** r_{x_m} **into** $M[t,o]$ **remove** Nr_{x_m} **from** $M[t,o]$
command $\mathit{revoke} \bullet r_{x_1}(u,t,o)$ **if** $M[u,o] = ar_1$ **then** **remove** r_{x_1} **from** $M[t,o]$ **enter** Nr_{x_1} **into** $M[t,o]$	**command** $\mathit{revoke} \bullet r_{x_1} \bullet \mathit{2}(u,t,o)$ **if** $M[u,o] = ar_2$ **then** **remove** r_{x_1} **from** $M[t,o]$ **enter** Nr_{x_1} **into** $M[t,o]$
$\vdots$	$\vdots$
command $\mathit{revoke} \bullet r_{x_m}(u,t,o)$ **if** $M[u,o] = ar_1$ **then** **remove** r_{x_m} **from** $M[t,o]$ **enter** Nr_{x_m} **into** $M[t,o]$	**command** $\mathit{revoke} \bullet r_{x_m} \bullet \mathit{2}(u,t,o)$ **if** $M[u,o] = ar_2$ **then** **remove** r_{x_m} **from** $M[t,o]$ **enter** Nr_{x_m} **into** $M[t,o]$
command $\mathit{grant} \bullet r_{d_1}(u,t,o)$ **if** $M[s,o] = ar_2$ **then** **enter** r_{d_1} **into** $M[t,o]$ $\vdots$ **command** $\mathit{grant} \bullet r_{d_n}(u,t,o)$ **if** $M[u,o] = ar_2$ **then** **enter** r_{d_n} **into** $M[t,o]$	

The system currently has some pending obligations where each obligation has the form $(\mathit{command},\ [t_s,\ t_e])$. Intuitively, the first ones nondeterministically select a truth assignment.

For each literal l_j^i we add one obligation. The time interval of this obligation depends on whether the literal is positive or negative, which assists in ensuring the obligations can be fulfilled during the cleanup phase.

$$\begin{gathered}
b_{1,\,t} = (\mathit{grant} \bullet r_{x_1}(u_1, u_0, o),\ [1,\ 2]) \\
b_{1,\,f} = (\mathit{revoke} \bullet r_{x_1}(u_1, u_0, o),\ [1,\ 2]) \\
\vdots \\
b_{m,\,t} = (\mathit{grant} \bullet r_{x_m}(u_1, u_0, o),\ [1,\ 2]) \\
b_{m,\,f} = (\mathit{revoke} \bullet r_{x_m}(u_1, u_0, o),\ [1,\ 2])
\end{gathered}$$

$$b_{l_j^i} = \begin{cases} (\mathit{grant} \bullet r_{d_i}(u_1, u_0, o),\ [3,\ 21]) & \text{if } l_j^i = x_k \\ (\mathit{grant} \bullet r_{d_i}(u_1, u_0, o),\ [3,\ 31]) & \text{if } l_j^i = \neg x_k \end{cases}$$

$$\begin{gathered}
b^* = (\mathit{grant} \bullet r'(u_1, u_0, o),\ [5,\ 31]) \\
b = (\mathit{grant} \bullet r_{goal}(u_1, u_0, o),\ [10,\ 15]) \\
b_{d_1}^{cleanup} = (\mathit{grant} \bullet r_{d_1}(u_2, u_0, o),\ [16,\ 20]) \\
\vdots \\
b_{d_n}^{cleanup} = (\mathit{grant} \bullet r_{d_n}(u_2, u_0, o),\ [16,\ 20]) \\
b_{x_1}^{cleanup} = (\mathit{grant} \bullet r_{x_1} \bullet 2(u_2, u_0, o),\ [16,\ 20]) \\
\vdots \\
b_{x_m}^{cleanup} = (\mathit{grant} \bullet r_{x_m} \bullet 2(u_2, u_0, o),\ [16,\ 20]) \\
b_{\neg x_1}^{cleanup} = (\mathit{revoke} \bullet r_{x_1} \bullet 2(u_2, u_0, o),\ [22,\ 30]) \\
\vdots \\
b_{\neg x_m}^{cleanup} = (\mathit{revoke} \bullet r_{x_m} \bullet 2(u_2, u_0, o),\ [22,\ 30])
\end{gathered}$$

Obligations $b_{k,\,t}$ and $b_{k,\,f}$ for $k \in [1,\ m]$ can always be executed before their deadline. Obligations $b_{d_i}^{cleanup}$, $b_{x_k}^{cleanup}$ and $b_{\neg x_k}^{cleanup}$ for $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$ can also be executed before their stipulated deadline enabling b^* and each $b_{l_j^i}$ to be carried out eventually. Now whether b can be carried out depends on how it is scheduled with respect to obligation b^*. If b^* is performed before b then b can not be executed because b^* adds u_0 to a right that prevents u_0 to receive the right granted in b. We show that S is satisfiable if and only if there exist a schedule in which b^* is performed before b; resulting the system being in weakly unaccountable state.

Now, suppose the 3-SAT problem is satisfiable, then we can construct a schedule that demonstrates the system is not weakly accountable. Such a schedule begins in period 1 by selecting an assignment that satisfies the 3-SAT problem, representing this in the assignment of each r_{x_k} to u_0 just in case x_k is true. Then, in period 2, u_1 grants each r_{d_i} to u_0, as permitted by statements (3.6) and (3.7). In period 3, u_1 grants r' to u_0, as permitted by statement (3.8). Finally, in period 4, u_1 is unable to fulfill obligation b due to statement (3.9)

Now, suppose there is no satisfying assignment for S. In this case, there is no scheduling of obligations $b_{1,t} \cdots b_{m,t}$ and $b_{1,f} \cdots b_{m,f}$ such that every $b_{l^i_j}$ can be performed before time 14. Thus, b^* can not be performed before time 15. This ensures b can execute in every schedule. The pending obligations at time 15 are then each $b_{l^i_j}$ and b^*. By time 20, u_2 must grant each r_{d_i} to u_0, enabling u_1 to fulfill b^*. Also by time 20, u_2 must grant each r_{x_k} to u_0, enabling u_1 to fulfill each obligation of form $(u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 21])$. Then, by time 30, u_2 must revoke each r_{x_k} from u_0 enabling u_1 to fulfill each obligation of form $(u_1\ Grants\ r_{d_i}\ to\ u_0,\ [3,\ 31])$. Thus all the obligations can be carried out, resulting the system being in a weakly accountable state. We can say that, when there is no satisfying assignment for S, the system is in weakly accountable state.

This reduction shows that checking weak accountability in a system of obligations governed by HRU Access Control Matrix Model policy is *co-NP Hard*. Now, to proof that the weak accountability procedure in a system of obligations governed by HRU matrix model is *co-NP Complete* we do the following. Given a schedule of obligations, for each obligation, we check if it is authorized. If it is authorized, we simulate the behavior of the obligation and update the authorization state of the system. Otherwise we check if the obligation is in its last time. If it is, we stop and declare the system is not weakly accountable. This can be done in $\mathcal{O}(n)$. Thus checking weak accountability in a system of obligations governed by HRU matrix model is *co-NP Complete*. □

3.4 Evaluation Results

A central goal of our empirical evaluation is to determine how practical it is to use the accountability algorithm as a part of a reference monitor. This section presents results of experiments designed to assess the adequacy of our algorithms and techniques with respect to performance. When a discretionary action is attempted, the problem that a reference monitor must solve is to determine whether the action, if permitted, would cause accountability to be violated. The discretionary action could do this by changing the current authorization state, causing new obligations to be incurred, or both. It is the performance of using our techniques to determine whether the discretionary action would lead to a violation of accountability that we wish to evaluate. When the determination must be made, there is an existing obligation pool, which in general includes new obligations that would be incurred if the discretionary action were permitted, and a current authorization state, which again reflects the state that would result if the discretionary action were carried out.

Our primary objective is to determine whether strong accountability checking can be added to the reference monitor. To this end, we evaluate strong accountability checking for two different authorization states, namely, mini-RBAC/mini-ARBAC and HRU Access Control Matrix Model. In our evaluation, we use a system with a moderate-size policy and 1000 users. For this, we perform two sets of experiments. The first evaluates the efficiency of the incremental algorithms

For an obligation set of size $100,000$, the algorithm runs in $5 - 7$ milliseconds when using the mini-RBAC/mini-ARBAC authorization state, and $70 - 90$ milliseconds when using the HRU Access Control Matrix Model as the authorization state. The second experiment is done on the non-incremental algorithms. On the same size input, it requires about 55 milliseconds under the mini-RBAC/mini-ARBAC authorization state, and 350 milliseconds using the HRU Access Control Matrix Model as the authorization state. The number of users and roles have little impact on the algorithms' execution times, and the effect of the number of policy rules is roughly linear. Thus we conclude that when the mini-RBAC/mini-RBAC authorization system is used, our algorithm for strong accountability provides adequate performance to be incorporated into reference monitors for most applications.

Another goal is to determine the effectiveness of all our approaches to weak accountability (when using mini-RBAC/mini-RBAC as the authorization state). We find that if the obligations overlap little and have a low degree of mutual dependence, the optimized special-purpose algorithm out-performs the model-checking approach; however, when obligations are clustered and have a high degree of mutual dependence, the model-checking approach tends to perform better. We introduce a metric called the Degree of OVErlapping (DOVE) that is the number of pairs of overlapping obligations, normalized with respect to the number of possible overlaps. Imagine a graph with nodes given by obligations and an edge connecting each pair of obligations that overlap in time. The DOVE is the size of the edge set divided by the number of edges the graph would have if it were complete.

Experimental Environment All the strong accountability experiments are performed using an Intel i7 2.0GHz computer with 6GB of memory running Ubuntu 11.10. Whereas, all the weak accountability experiments are performed using an Intel Core 2 Duo 2.0GHz computer with 2GB of memory running Ubuntu 8.10. The algorithms for strong and weak accountability are implemented in C++ and built with gcc 4.2.4.

3.4.1 Evaluation of the Strong Accountability Algorithm

This section presents the policy and obligations generation for both the mini-ARBAC/mini-RBAC and the extended mini-HRU Access matrix control authorizations. As well, the empirical evaluations for the strong accountability algorithms presented in previous sections.

mini-ARBAC/mini-RBAC To evaluate the strong accountability algorithms for mini-ARBAC/mini-RBAC, we assumed 1000 users and used a handcrafted mini-ARBAC/mini-RBAC policies $\langle \gamma_0, \psi_0 \rangle$ summarized in table 3.2. To generate the obligations, we handcrafted 6 strongly accountable sets of obligations in which each set has 50 obligations. Each set has a different ratio of administrative to non-administrative obligations (rat). We then replicated each set of obligations for different users to obtain the desired number of obligations. The execution times shown are the average of 100 runs of each experiment.

Table 3.2: Policies used in experiments. C represents the number of roles in the pre-conditions of CA rules

policy	R	O	A	CA	CR	PA	C	Generated
$\langle \gamma_0, \psi_0 \rangle$	50	50	50	60	60	250	10	By Hand
$\langle \gamma_1, \psi_1 \rangle$	50	50	50	2500	2500	120,000	10	Random
$\langle \gamma_2, \psi_2 \rangle$	27	12	37	37	38	1,200	4	By Hand

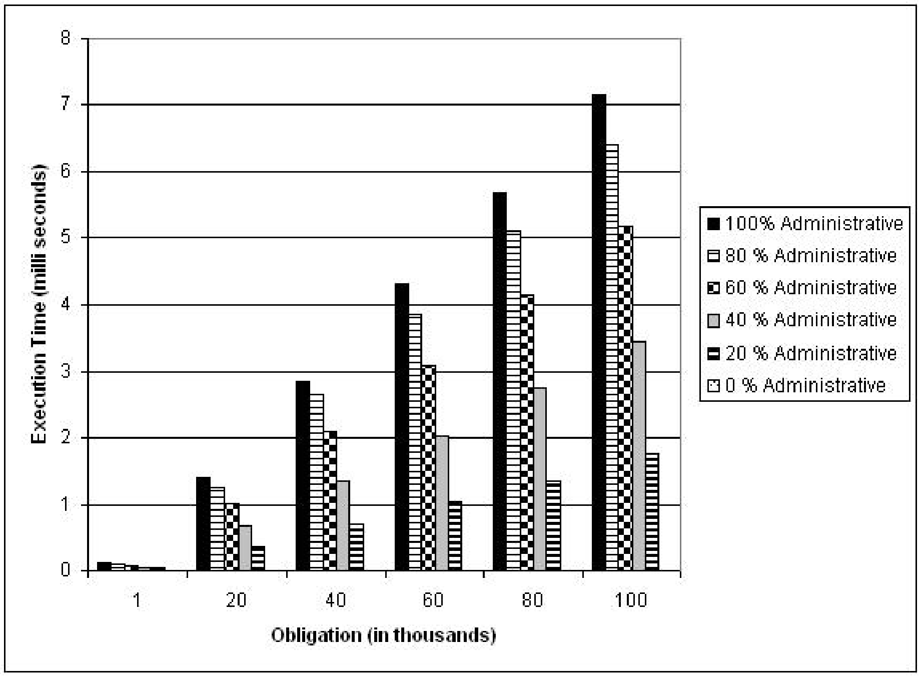

Figure 3.3: Performance of incremental strong accountability algorithm - mini-RBAC/mini-ARBAC

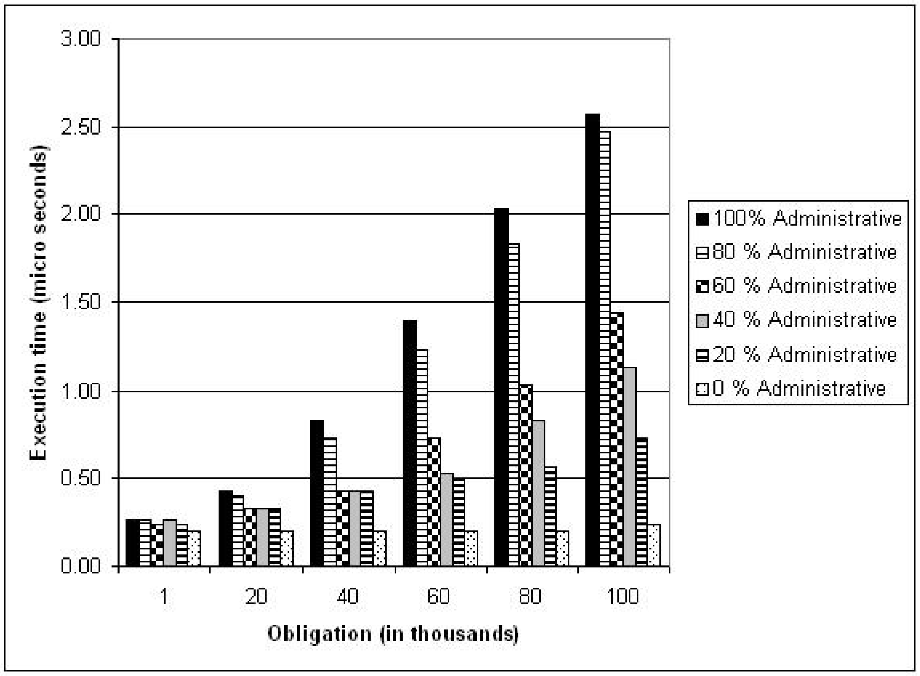

Figure 3.4: Performance of incremental strong accountability algorithm - mini-RBAC/mini-ARBAC (non-strongly accountable set)

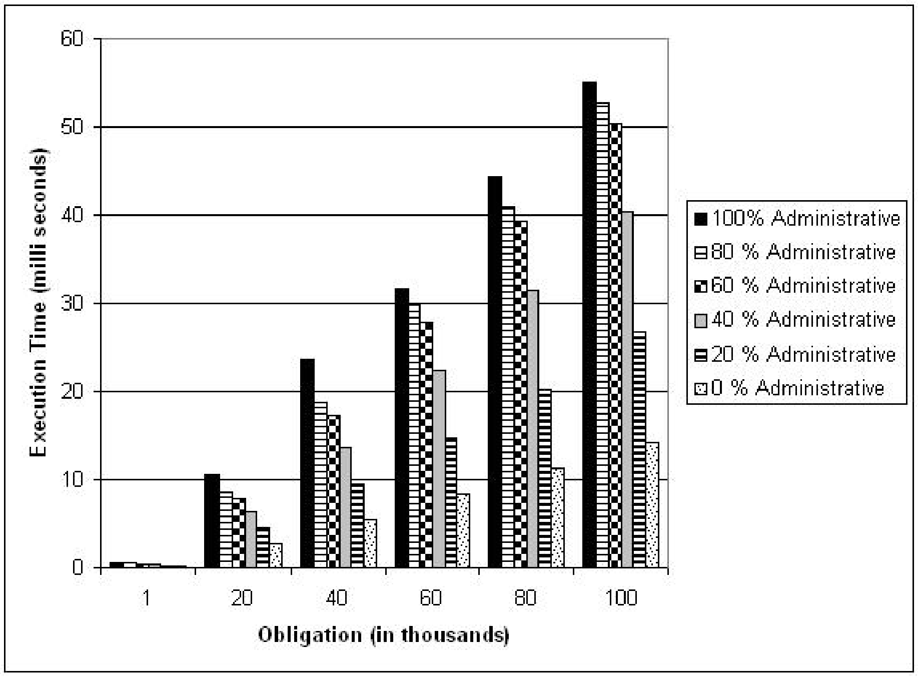

Figure 3.5: Performance of non-incremental strong accountability algorithm - mini-RBAC/mini-ARBAC

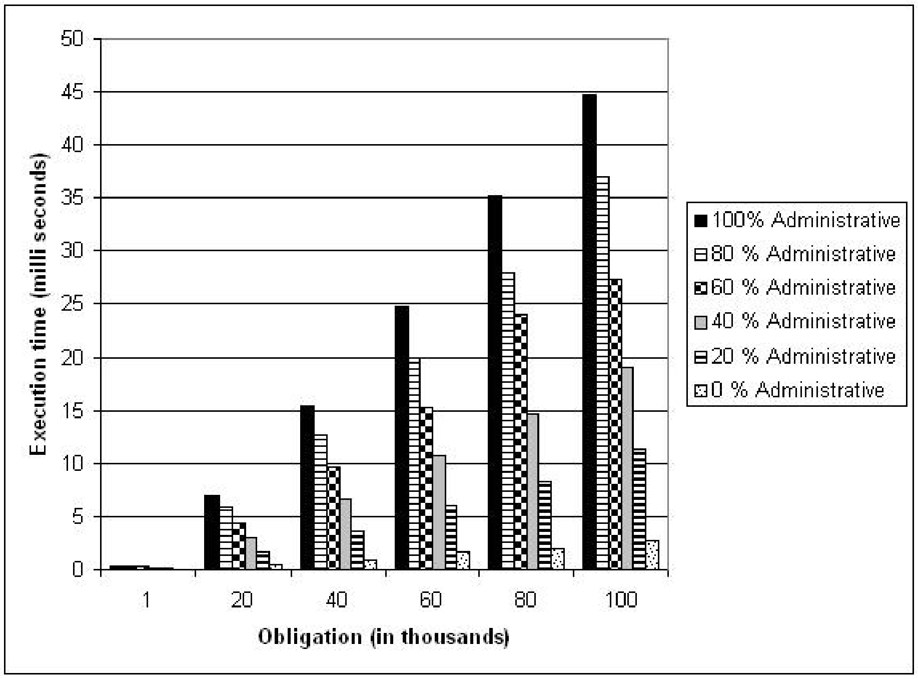

Figure 3.6: Performance of non-incremental strong accountability algorithm - mini-RBAC/mini-ARBAC (non-strongly accountable set)

Figure 3.3 presents results for the incremental algorithm. As we can see, the time required by the incremental algorithm grows roughly linearly in the number of obligations. The impact of rat on the execution time of algorithm arises largely because the algorithm must inspect every obligation following each administrative obligation. On the other hand, when someone tries to add a unauthorized obligation to a strongly accountable set, the algorithm can determines that the final set is not accountable in less than 2.5 micro-seconds as shown in figure 3.4. Recall that the incremental algorithm first checks whether a new obligation is authorized or not. This is made by using a modified search tree.

In the experiments for the non-incremental algorithm (SA), we see that the execution time grows roughly linearly with the size of the obligation set (figure 3.5). As with the incremental algorithm, a higher rat value leads to a higher execution time. Here, the impact of having a non-strongly accountable set is much greater than in the incremental case. Figure 3.6 presents this case. In the worst case, the algorithm could determine that an set if not strongly accountable in 45 milliseconds. The time to determine that an obligation set is not strongly accountable will be less than to determine if a set is strongly accountable, because as soon as the system finds one obligation that is not authorized it can terminate the algorithm.

Extended mini-HRU Access Control Matrix Model To evaluate the strong accountability algorithm for the extended mini-HRU Access Matrix model, we also assume 1000 users and used a handcrafted extended mini-HRU policy that contains 60 objects, 250 non-administrative commands, 120 administrative commands, 50 rights. The maximum number of preconditions of non-administrative and administrative is 10. Again, we use the same handcrafted strategy that was used when generating sets for the mini-ARBAC/mini-RBAC.

Figures 3.7 and 3.8 show the simulation results of the incremental strong accountability algorithm when using the extended mini-HRU as the authorization state. Note that, the results presented for this algorithm behave as the results presented in figure 3.4 and 3.6. However, because the extended mini-HRU authorization state supports multiple changes on the authorization

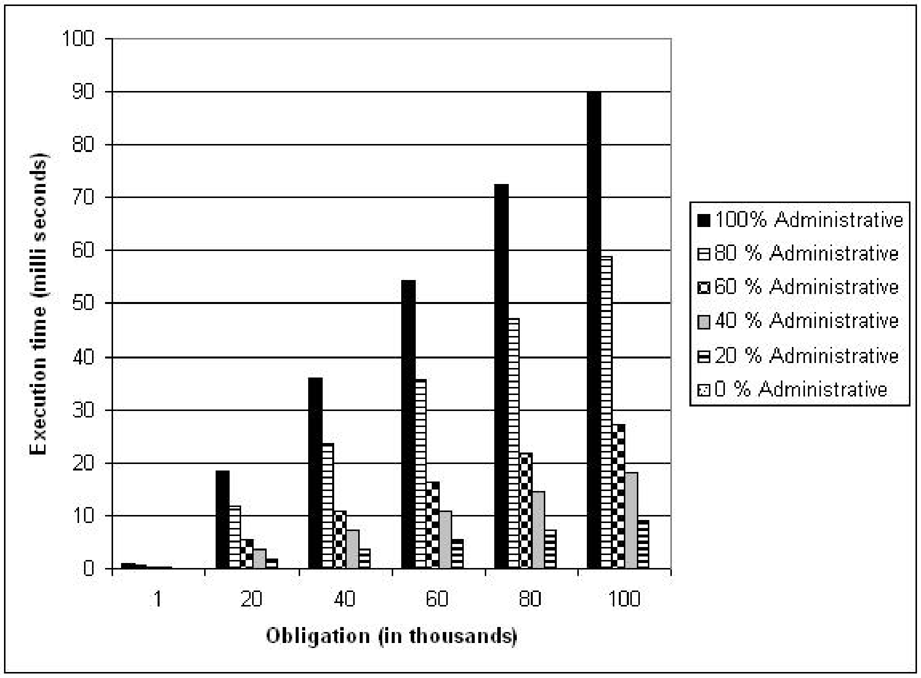

Figure 3.7: Performance of incremental strong accountability algorithm - HRU

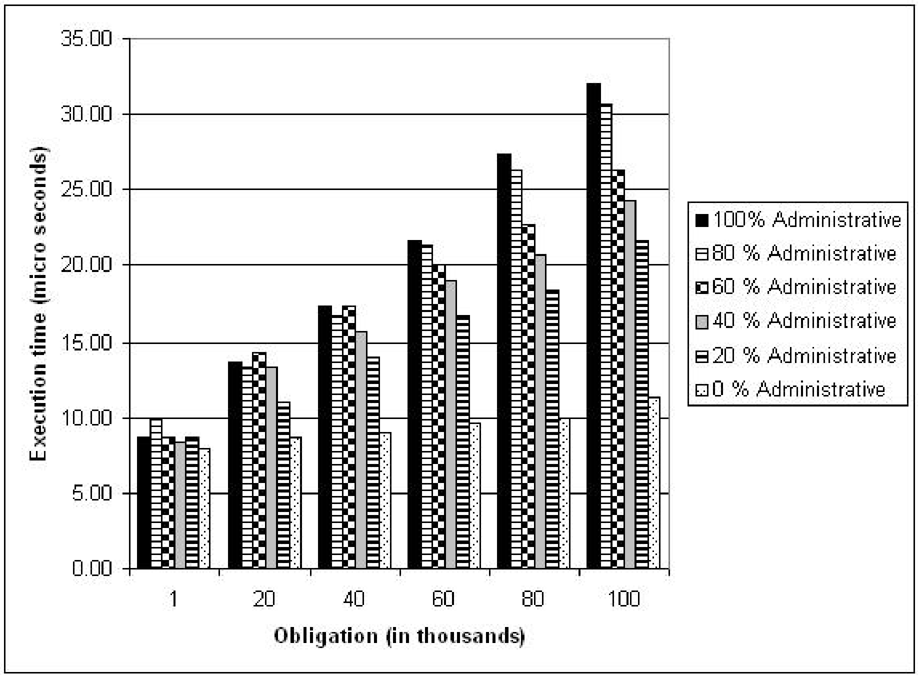

Figure 3.8: Performance of incremental strong accountability algorithm - HRU (non-strongly accountable set)

per action, the cost to decide accountability is greater than that of deciding it when considering the mini-RBAC/mini-ARBAC as authorization state. In our experiments, deciding strongly accountability when using extended mini-HRU is 11 times slower than when deciding it using the he mini-RBAC/mini-ARBAC as authorization state.

Figures 3.9 and 3.10 show the empirical evaluation for the non-incremental strong accountability using the extended mini-HRU model as the authorization state. Again, this algorithm behaves in the same fashion of the non-incremental strong accountability algorithm when using mini-RBAC/mini-ARBAC as authorization state. In our experiments, the HRU version is 7 times slower than the mini-RBAC/mini-ARBAC version.

3.4.2 Evaluation of the Weak Accountability Approaches

In this section, we compare the performance between the special-purpose algorithm (SP), the optimized special-purpose (OSP) algorithm, and the model checking approach (MC). We used two policies, $\langle\gamma_1, \psi_1\rangle$ and $\langle\gamma_2, \psi_2\rangle$, summarized in table 3.2. (See [7] for the detailed policies.) To generate obligations, we used two algorithms, A_1 and A_2. (See [7] for more details.) A_1 uses policy $\langle\gamma_1, \psi_1\rangle$ to generate a random set of obligations for given values of DOVE, number of obligations (n), and rat. A_2 uses policy $\langle\gamma_2, \psi_2\rangle$ and the parameters presented above, and generates obligations like in section 3.4.1. In the experimental results presented in table 3.3 and 3.4, the entry x means the approach was unable to terminate within a reasonable time (60 minutes) for the input set.

Table 3.3: (a) Execution time of the MC, SP and OSP approach vs. the number of obligations (n). (b) Execution time of the MC, SP and OSP approach vs. DOVE.

n	100	300	700
MC	1.0	16.1	324.8
SP	0.006	0.025	1.42
OSP	0.002	0.003	0.016

(a) Time in seconds

DOVE	0.02	0.06	0.19
MC	0.80	0.85	0.83
SP	0.001	179.0	x
OSP	0.0006	0.0006	0.0006

(b) Time in seconds

Table 3.3 (a) shows the execution time vs. number of obligation(n) for MC, SP and OSP. Obligations were generated by A_1 using DOVE =0.002 and rat = 5%. The execution time for MC

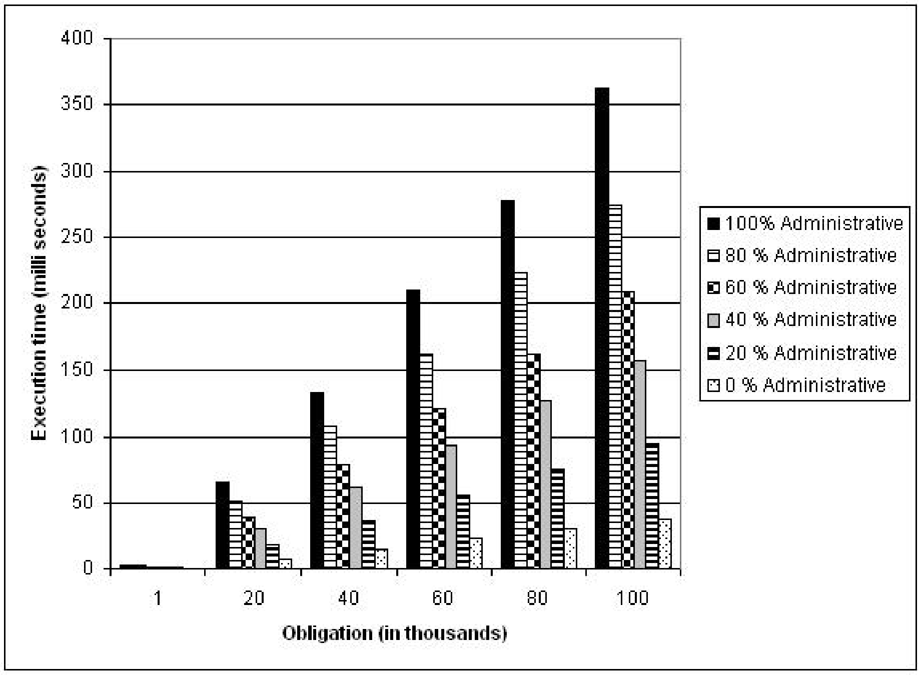

Figure 3.9: Performance of non-incremental strong accountability algorithm - HRU

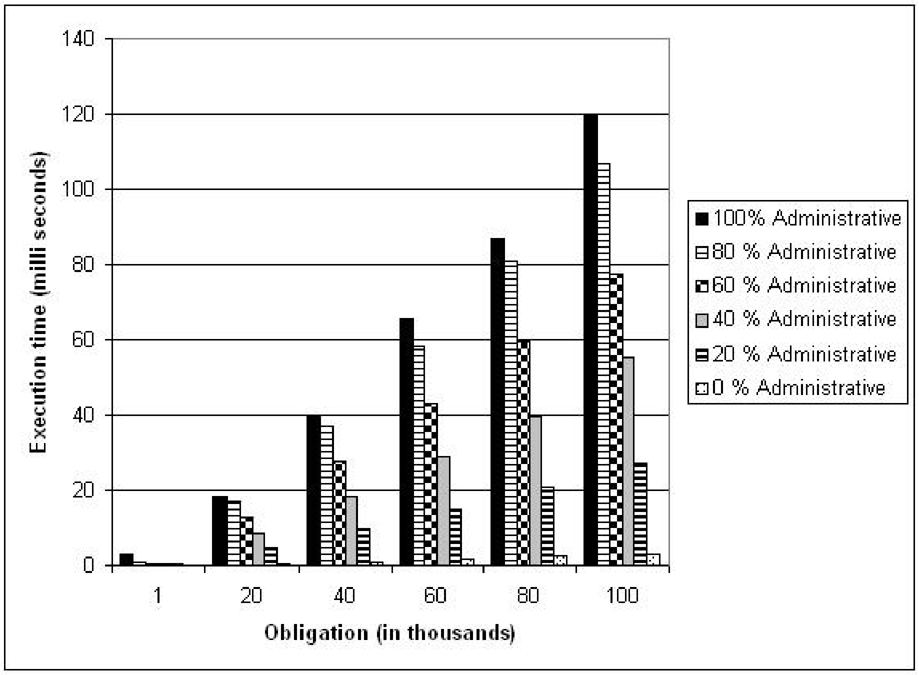

Figure 3.10: Performance of non-incremental strong accountability algorithm - HRU (non-strongly accountable set)

grows rapidly with n, as this and the number of users increases the size of the state space. On the other hand, execution time of SP and OSP grows slowly with n and OSP has a much better performance than SP.

Table 3.3 (b) shows execution time vs. DOVE for MC, SP and OSP. We used 100 obligations generated by A_2 with a rat value of 20%. The execution time of MC and OSP are not affected by DOVE, whereas that of SP grows exponentially. We empirically determined that the value of DOVE 0.158 forms a threshold for SP beyond which it does not perform adequately. Again, the execution time for the OSP approach is much faster than the MC approach.

Table 3.4: (a) Execution time of the MC, SP, OSP approach vs. different rat value. (b) Execution time of the MC, SP, SA and OSP vs. n.

rat	0%	10%	20%
MC	97.9	99.3	104.1
SP	91.1	92.8	96.8
OSP	0.004	0.004	0.005

(a) Time in seconds

n	10^3	10^4	3×10^4
MC	1831.5	x	x
SP	0.003	0.055	1.859
OSP	0.008	0.618	5.27
SA	0.001	0.006	0.052

(b) Time in seconds

Table 3.4 (a) presents execution time vs. rat for all the techniques using A_2 where DOVE = 0.01 and $n = 500$. The rat affects MC and SP techniques' execution time, whereas the OSP does not seem to be affected and has a speedup of magnitude 25,000 compared to both MC and SP. In our experiments, we used A_2 to generate obligations for most of our inputs, as it is difficult in practice to randomly generate large sets of obligations that are weakly accountable.

In our experiments, we found problem instances where the MC approach outperforms the OSP approach (*viz.*, when the obligations are clustered together and are dependent on each other) and vice versa. We use this fact and introduce a hybrid approach (see figure 3.11). In this approach, we use the fact that strong accountability implies weak accountability so we check if the obligation set is strongly accountable. If not, then we perform some quick tests, which can identify some of the problem instances that are definitely not weakly accountable. We present the tests below:

1. We simulated the schedule in which each obligation is executed at its end time. If under this

schedule any obligation is not authorized, the obligation set is not weakly accountable. (The converse is not true because other valid schedules may also provide counter examples.)

2. If some obligation b requires a user to be in a role r and another obligation b' that overlaps with b with $b.\mathrm{end} \leq b'.\mathrm{end}$ and that revokes u the role, then the set of obligations is not weakly accountable. (No third obligation could restore the role, should b' be executed before b.) Similarly, if b requires the user not to occupy the role, and b' grants it, the set is not weakly accountable.

If the test is unable to decide the accountability query, then we compute two metrics: DOVE and dependency; based on these metrics we choose whether we use the MC or OSP approach. In case that both metrics are unsatisfied, we execute both approaches. If neither of the approaches terminates within a specified amount of time, we say that the problem instance can not be solved.

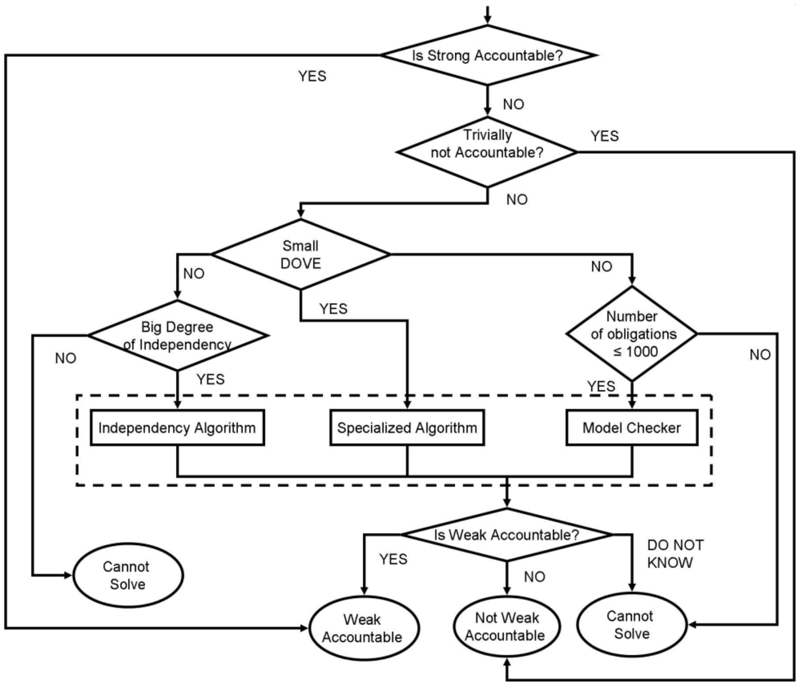

Figure 3.11: The Hybrid Approach for solving Weak accountability

3.5 Summary

In this chapter, we have presented an algorithm to decide strong accountability efficiently and two methods to decide weak accountability. These are based on an obligation system that uses mini-ARBAC/mini-RBAC as its authorization system. We have given experimental results that demonstrate that the performance of the algorithm for strong accountability is excellent and that show the methods for weak accountability are adequate for most medium-size problem instances. This is despite our result that, even using the simple authorization model, mini-ARBAC/mini-RBAC, the weak accountability problem is co-NP complete with respect to the policy and obligation pool size. In addition, we have also presented an algorithm for deciding strong accountability for a obligation system that uses extended mini-HRU Access Matrix Model as authorization system. In general, this algorithm performs 13 times slower than the mini-ARBAC/mini-RBAC based model. Despite on this, it still runs in less than 90 milliseconds.

CHAPTER 4: FAILURE FEEDBACK

In previous chapters, we presented techniques to decide accountability efficiently in practice. When such approach is used incrementally, one can expect that the system will preserve accountability. However, many situations can arise that can cause the violation of accountability property of the system. Sometimes a user is fired or transferred, a server can crash, and these situations can put the system in an unaccountable state. We intend to design automated tools that will enable the administrators and other authorities to remove or reassign obligations, or to add new obligations that replace old ones to restore accountability. In prior chapters, we did not address the relationship of usability and security. Addressing the relationship between security and usability is the main goal of this chapter.

In recent years, several researchers have proposed techniques for providing users with assistance in understanding and overcoming authorization denials [36, 57, 58]. The incorporation of environmental factors into authorization decisions has made this particularly important and challenging. An environmental factor that has not previously been considered in this effort to provide such assistance to users arises in systems, where obligations can depend on and affect authorizations. In these systems, it is desirable to ensure that users will have the authorizations they require to fulfill their obligations, so, when desired actions are denied, the system provides to the users a plan of actions to help them find means of overcoming their denials. We call the approach to present a plan of actions to the users, the *Action Failure Feedback Problem* . The approach to solve this problem was developed by Irwin in his PhD dissertation [30]. The contributions of my dissertation to this topic are a precise definition of the problem, the complexity analysis of the problem, and empirical evaluations to demonstrate the effectiveness of the approach presented in [30]. Since obligation systems of the kind we study are not yet deployed, it is impractical to obtain real policies and obligations. Therefore, the problem of generating representative problem instances is a significant challenge. In this chapter, we discuss a three-fold approach we took to generate what we believe to be a reasonably representative collection of problem instances upon which we based our empirical performance evaluations on.

4.1 Action Failure Feedback Problem

An *instance of the action failure feedback problem* (*AFFP*) is given by a tuple $\langle \gamma_0, \psi, U_0, u_t, A, B \rangle$ in which:

- $\gamma_0 = \langle R, \mathit{UA}_0, \mathit{PA} \rangle$ is an initial mini-RBAC authorization state.
- $\psi = \langle \mathit{CA}, \mathit{CR}, SMER \rangle$ is a mini-ARBAC*. Here, differently from the mini-ARBAC version presented in chapter 2, we are considering $SMER \subseteq R \times R$ (Static Mutually Exclusive Roles) rules, that are unordered pairs of roles such that no user is allowed to have both roles simultaneously. Our *CR* rules are also conditional, in other words, like the *CA* the target users must satisfies the preconditions.
- U_0 represents the set of users in the system.
- u_t is the user that intends to perform the actions in A.
- B is a set of strongly accountable pending obligations.
- A is a set of desired actions that we want to add to B while preserving strong accountability. These actions must be performed by u_t. The actions are defined as a set of *desired_action* that can have one of the following formats:
  1. For non-administrative actions, we have $\langle u, a, o \rangle$, where a user u intends to perform non-administrative action a on object o.
  2. In the case of administrative actions, we have $\langle u_a, grant, r, u \rangle$, in case of a Grant, meaning that user u_t intends to grant role r to user u. Or, $\langle u_a, revoke, r, u \rangle$, when user u_t intents to revoke role r of user u

Definition 20 (Action Failure Feedback Problem)**.** *Does there exist a set of actions A_1 and an assignment of time periods to them, where $A \subseteq A_1$ and $B \cup A_1$ is strongly accountable ?*

4.1.1 Complexity of the Problem

In this section, we discuss the complexity of the action failure feedback problem. We show that the problem is PSPACE-hard with respect to the policy and obligation pool size. To show that the problem is PSPACE-hard, we reduce a well known problem called the unrestricted role reachability problem [53] to the action failure feedback problem.

A user-role reachability problem instance is a tuple $Rea = \langle \gamma_0^R, \psi^R, U_0^R, u_t^R, goal^R \rangle$. In the tuple, $\gamma_0^R = \langle R^R, UA_0^R \rangle$ is an initial mini-RBAC authorization state, $\psi^R = \langle CA^R, CR^R, SMER^R \rangle$ is a mini-ARBAC* policy, U_0^R is a set of users, $u_t^R \in U_0^R$ is a target user, and $goal^R$ represents a set of roles in the system, $goal^R \subset \gamma_0^R.R$.

Definition 21 (Role Reachability Problem)**.** *Is it possible for the administrative users in U_0^R to grant u_t^R all the roles defined in the goal by just applying the rules on ψ^R ?*

4.1.1.1 Reduction

We want to reduce the *Rea* problem into the *AFFP* problem. So, the *AFFP* instance will be:

- $u_t = u_t^R$ and $\psi = \psi^R$
- $B = \varnothing$ and $U_0 = U_0^R \cup \{u_t^R\}$
- $UA_0 = UA_0^R$
- $\gamma.PA = \{\langle r, \langle a, o \rangle \rangle | (r \in goal^R) \wedge (a \in new\_action()^1) \wedge (o \in new\_object())\}$ (*i.e.*, for each role presented in the goal, we create a new entry in PA, where this new entry is a unique permission).
- $A = \{\langle u_t, a, o \rangle | \exists r \in R . \langle r, a, o \rangle \in \gamma.PA\}$. We create one desired action for each new PA entry that was added in the previous step.

[1]The methods new_action(), new_object() return a unique new action and a unique new object respectively.

The intuition behind the reduction is that for each role $r_i (0 \leq i \leq |\text{goal}|)$ in goal we create a new unique desired action A_i in *AFFP*. We also add the associated PA rules which permit a user in role r_i to perform action A_i. In addition, we assume no pending obligations. The reduction can clearly be done in polynomial time. Thus, if we find a solution for the *AFFP*, we also find a solution for the *Rea* problem.

4.2 Approach

In this section, we summarize the approach taken by Irwin *et al.* [30] for solving the AFFP. As presented in section 4.1.1, the *AFFP* is PSPACE-hard, then there is no algorithm to solve it in polynomial time. Partial-order planners are search space engines that use heuristics techniques. Thus, partial-order planners are an inherent choice to attack our problem.

A typical partial-order planner has a list of ordering constraints, a set of actions, a set of constraint variables, and a list of flaws that contains conflict among the actions, and the unsatisfied goals. Initially, the planner starts with a empty initial plan. When it progresses, it chooses a flaw and tries to fix it by creating additional new steps (*e.g.*, ordering constraints, variable constraints, etc.), and it also updates the list of flaws. The planner stops when it finds a plan without any flaws (found a solution for the given problem), or when pre-defined threshold of time has passed (could not find a valid solution for the problem instance).

We use a modified version of the UCPOP [44] partial-order planner for finding solutions of AFFP instances. This choice is based on the reliability of the UCPOP, and a good support of the AI community. The following discusses how we convert an AFFP instance Action Description Language (ADL) which is the input language of the planner, as well as some modifications we make for adapting UCPOP for our purpose.

1. **Translation and Initial Plan:** We first translate the AFFP problem instance into ADL. In this translation, actions in the access control system become actions in the planning domain, carrying appropriate preconditions and effects. We also, create a initial plan, which contains

the set of pending obligations and the desired actions.

2. **Preconditions and Flaws:** Because we modify the planner to take an initial plan that includes both desired actions and obligations, we must explicitly construct the list of flaws for the planner that includes the preconditions for all the desired actions.

3. **Variable Constraints:** In a partial-order planner, arguments taken by actions are represented by variables. The variables occurring in obligations (currently in the pool) and the new desired actions should be constrained so that their values are uniquely determined.

4. **Timing:** In order to encapsulate obligation into a partial-order planner we need to translate the discrete times of the obligations into a partial-order relation that is used in the initial plan.

5. **Converting from Planner Output to Obligations:** After a planner gives a plan of action, we need to convert the planner output into obligations. The intuition behind the conversion it to use a topological sort algorithm to convert the partial order of obligations provided by the planner to a possible total order of obligations.

4.3 Evaluation of the Planner

In section 4.1.1, we have shown that the action failure feedback problem is PSPACE-hard. This has lead us to investigate using AI planning techniques that are well known to solve search-space based problems using heuristics. This section presents our empirical evaluation of the effectiveness this approach.

The main goal of our empirical evaluation is to assess the limits of our approach for small size, but complex problem instances. To our knowledge, there are currently no deployed systems that incorporate the support of management and enforcement of obligations as part of their security policy. Consequently it is necessary for us to devise our own policies and problem instances. We have three classes of input instances; two are generated from automated input instance generators and the third is hand crafted. We discuss our problem instance generation techniques in one of the following sections. After presenting the equipment and software employed, this section discusses our approach to generate those problem instances, followed by our results and findings.

4.3.1 Experimental Environment

All the experiments here are performed using an Intel Core 2 Duo 3.0 GHz computer with 2 GB of memory running Ubuntu 9.04. One of the input instance generators is written in C++. The other input instance generator and the translator which translates a mini-RBAC and mini-ARBAC* policy to ADL is written in Objective Caml. We used UCPOP 4.1.

4.3.2 Problem Instance Generation

In an effort to evaluate our planner-based approach as fully as possible, we used three different approaches for generating problem instances. All three approaches generate problem instances that include obligation pools that are known to be accountable and desired actions for which it is known that plans that enable the desired actions to be added to the pool, along with other enabling obligations, resulting in a new pool that is again accountable. The only question is whether the planner is able to find the enabling actions to achieve this result.

We label the three test generators used G_1, G_2, and G_3, respectively. The goal of G_1 is to generate policies, obligation pools, and desired target actions randomly based on uniform probability distributions for each component of the problem instance. The goal of G_2 is to generate instances that seem likely to arise in a realistic system deployment setting withing an enterprise. The goal of G_3 is to generate instances that are particularly intricate with the goal of making the discovery of solutions particularly difficult for the planner. Instance generator G_1 and G_2 are fully automated, while approaches G_3 generates instances manually.

The automated generation of instances by G_1 is based on certain input parameters. Specifically, these are the number of roles (r), the number of pre-conditions in the CA rules (c), the number of CA rules (*ca*), number of CR rules (*cr*), number of users (u), number of roles per user (*ru*), number of actions (a), the number of objects (o), the number of PA rules (*pa*), number of obligations (*obl*) in the existing obligation pool, number of maximum length of the sequence of new actions that need to be added to enable the desired target actions to be enabled (*depth*), and number of desired target actions (*dact*). The CA rules, CR rules, $\gamma.U$, $\gamma.O$, $\gamma.PA$ and $\gamma.UA$ are generated randomly based on the parameters ca, cr, u, o, a, o, and ru, respectively. For generating each obligation in the obligation pool, we randomly select a time window and try all possible obligations that make the system strongly accountable. We select one randomly from the possible obligations and add it to the pool of pending obligations (B). This process is repeated until a pool of size *obl* is obtained. For generating desired target actions, we first simulate the effect of administrative obligations of B initial authorization state γ to get a new authorization state γ_1. We then iteratively generate all administrative actions that are authorized in the current authorization state, select one at random and add it to a plan that is constructed to ensure the desired target action can actually be achieved. (It is a plan such as this that the planner will be expected to find.) The authorization state is then updated accordingly. This process is repeated, each time selecting randomly among administrative actions that are currently authorized, but that would not have been authorized in the previous authorization state, until a plan of the desired length is reached (*depth*). The final action is then generated, again selected at random among actions that are authorized in the final authorization state, but not in the previous one. Due to the cost of this algorithm, we found it necessary to limit *depth* to at 5. In our experiments, G_1 is used with *ca* = 100, *cr* = 100, *pa* = 200, and *u* = 40.

For the second test generator (G_2), we automatically constructed a system state intended to be reflective of a realistic enterprise organization structure. For that, we use one level of administrative roles (10 roles), and three levels of operational roles, 5, 10, and 20 roles, respectively for each level. For G_2, we have a total of *u* = 180, where 40 are administrative users, and 140 are operational users. For each role, we have two can_assign rules and two can_revoke rules, in a total of *ca* = 90 and *cr* = 90. For each obligations have 20 units of time window, and the time frame for the set of all obligations is 200.

For the third test generator (G_3), we use a handcrafted policy based on a hospital setting with r = 58, c = 6, *ca* = 98, *cr* = 55, *pa* = 168, *obl* = 60 and *dact* $\leq$ 20. We design complex problem instances (*i.e.*, when desired actions interfere with other desired actions, obligations interfere with desired actions, desired actions interfere with obligations, and also considering Statically Mutually Exclusive Roles ($SMER$)). We also generate inputs where authorizing each desired action needs introducing more than 5 additional actions in the plan.

4.3.3 Results

The execution times reported here are an average of 10 successful runs. In our experiments, we put a bound on the number of possible plans the planner can examine. In addition, we put a time threshold in which the planner can inspect for solutions. Thus, when the planner cannot find a solution in the assigned threshold time; it considers the problem instance it is inspecting does not have solution (*i.e.*, we consider this a failure). We have 18 failures in a total of 403 trials for a success rate of 95.53%.

We use our third input instance generator (G_3) to identify the limits of our approach. We observe that our approach is adequate for problem instance with *dact* = 1 and *obl* = 60, when the optimal plan length is 7. This signifies that authorizing the desired action needs introducing 6 additional actions. Our approach does not scale well (greater value of *dact*) for such complex problem instances.

Table 4.1 shows the execution time (in seconds) of problem instances generated by G_1 ($Time_{G_1}$) and G_2 ($Time_{G_2}$) with varying *obl* values. All the other parameters are fixed. The execution time grows linearly with the number of total actions for both generators. As, G_1 generates complex problem instances, $Time_{G_1} \geq Time_{G_2}$. When the total number of actions exceeds a certain threshold (170 actions), the execution time starts growing exponentially.

Table 4.2 (a) reports execution time with varying c (*dact* = 10, *obl* = 80). The execution time increases linearly when $c < 10$ but it starts growing exponentially for $c \geq 10$. Table 4.2 (b) shows execution time for different *dact* values (*obl* = 80). The execution time in this experiment increases linearly with *dact*. G_1 is used to generate input instances for both the experiments.

Table 4.1: Execution time vs. *obl*

dact	***obl***	$Time_{G_1}(s)$	$Time_{G_2}(s)$
10	10	4.03	1.19
10	20	7.43	2.46
10	40	14.69	6.87
10	80	55.89	24.24
10	160	304.64	152.81

Table 4.2: (a) Execution time vs. c (b) Execution time vs. *dact*

c	***dact***	***obl***	$Time(s)$
1	10	80	56.29
3	10	80	60.59
5	10	80	66.70
8	10	80	64.61
10	10	80	116.94

(A)

dact	***obl***	$Time(s)$
5	80	45.48
10	80	48.98
15	80	62.29
20	80	69.05
30	80	82.29

(B)

4.4 Summary

In this chapter, we have presented the formal definition of the action failure feedback problem. Such problem attempts to provide suggestions to users how they could accomplish desired target actions that are currently denied. Our formulation of the problem is based on user obligation systems that use mini-RBAC and mini-ARBAC* as their authorization system. We have shown that the action failure feedback problem is PSPACE-hard with respect to the policy and obligation pool size, and we have summarized the AI planner-based [30] approach to solve it. We have constructed a tool based on this approach and evaluated it empirically on a diverse collection of problem instances. The results demonstrate that our approach could be useful for small size problem instances.

CHAPTER 5: RESTORING ACCOUNTABILITY

As we have discussed in previous sections, a reference monitor can help maintain accountability by preventing actions that would cause it to be violated. However, even with such a reference monitor in place, accountability is still violated when an obligation is not or will not be performed. For instance, if a user fails to fulfill an obligation, say, to grant Alice the rights she needs next July, the system will become unaccountable, and Alice will be unable to perform her own obligation. Thus, an obligation system manager needs strategies and support tools that she can use to restore accountability. Three of the four present contributions seek to address these needs.

As part of supporting an obligation system manager in restoring accountability, we present three forms of dependency that can exist among obligations within a system's obligation pool. While functional dependencies also exist, here we focus exclusively on dependencies that are based on authorization requirements. The three kinds of authorization dependencies we formalize are positive dependency, negative dependency, and antagonistic dependency.

Borrowing a term from programming languages, we introduce what we call a slice of an obligation pool. A slice of a program is a subset of the statements in a program that define a portion of the program's behavior [56]. In our context, a slice is a subset of the current pool of pending obligations. We introduce two kinds of slice. One is based on positive dependency among obligations. The other is based on all three forms of dependency mentioned above. An obligation system manager who is working on restoring accountability can use a slice of the current obligation pool to identify which obligations she needs to consider modifying. As we shall see, the choice of which kind of slice to use depends on the strategy being applied to the restoration of accountability.

Our final contribution consists of several strategies that an obligation system manager can use for accountability restoration. These strategies can be supported by AI planning techniques [47] and by tools that compute the kinds of obligation pool slice discussed above.

5.1 Obligation Dependencies

Our system uses a reference monitor that attempts to maintain accountability by denying action requests that would violate it. Accountability can be violated nevertheless. For instance, suppose that obligations b_1 and b_2 are scheduled so that b_1 happens first and grants necessary permissions for performing b_2. If b_1 is violated, b_2 may no longer be authorized, so the obligation pool ceases to be accountable. A similar situation arises when a manager or administrator learns ahead of time that, if nothing is done to prevent it, b_1 will become violated owing to user or resource unavailability. As the violation is anticipated before it has occurred, in this latter case it may be easier to recover gracefully than in the former. In both cases, however, methods for restoring or preserving accountability are needed.

Ultimately, when accountability is violated, a human obligation system manager will generally have to participate in its restoration. To facilitate her task, we can provide information about dependencies among obligations that are relevant to the various approaches available to her for this purpose. For instance, in the previous example, b_2 cannot be fulfilled if b_1 is violated, as b_1 provides the necessary permissions for b_2. When an obligation is or will be affected by the (eventual) violation of another obligation, we say that the former has a dependency on the latter obligation. Changes in the obligation pool that affect accountability can have their impact on other obligations indirectly. The obligation system manager is best served by being provided an aggregation of dependencies that connect multiple obligations to the source of the disruption of accountability. We call the aggregate we define for this purpose an *obligation-pool slice*. In this section, after discussing each form of dependence, we return to the notion of a slice.

Various forms of dependence arise depending on the strategy one tries to use to restore accountability. In this section, we identify three different categories of dependencies on this basis. When one plans simply to let an obligation go unfulfilled ("removing" the obligation), then one is concerned only with the impact this will have on later obligations via the authorization state. We call this *positive dependence*. (Positive dependence can also arise when the violated obligation

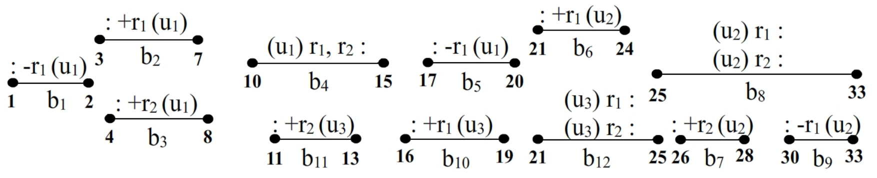

Figure 5.1: Dependencies among obligations

revokes a role if the second obligation is an administrative action that requires the target user not to have the role.) When one plans to reschedule an unfulfilled obligation, it may be necessary also to reschedule later obligations that have a positive dependence on it. When this is done, the second obligation is moved later in time, possible moving it past some other obligation that modifies the authorization state in a way that interferes with the execution of the second obligation. We call this interference *negative dependence*. Finally, when one obligation is rescheduled to occur after an obligation it once preceded, the authorization state in which a third, later obligation must be executed may be different as a result. For instance, if an obligation is moved past another that would reverse a change made by the first to the authorization state, the third obligation would no longer be exposed to an authorization state in which the reversal has been applied. We call this third form of dependence *antagonistic dependence*.

The notions of dependency among obligations presented here are based only on the authorization requirements of the obligations. There could be other notions of dependency, such as functional dependencies that expresses requirements on the temporal ordering of obligations. However, we do not consider this here.

Example 22 (Obligations)**.** *We use the obligation pool pictured in figure 5.1 to illustrate the various notions of dependencies. Each obligation has a unique identifier, b_i. The roles r_j that the obligated user requires to perform his obligation are given on the left hand side of the colon, along with the obligated user u_k. In this example u_1 executes b_4, u_2 executes b_8, u_3 executes b_{12}, and u_0 is responsible for the rest. When multiple policy rules enable the obligation, the requirements for each rule are presented in separate annotations, as illustrated by obligation b_8. When one rule requires multiple roles, they are listed in the same annotation, as illustrated by obligation*

b_4. *For economy of space, we have chosen an example that avoids the need to represent negative preconditions. When an obligation performs an administrative action (grant or revoke), the right hand side of the colon indicates the effect of this action. When an obligation grants a role* r *to a user* u*, this is indicated by* $+r(u)$ *on the right hand side of the colon; when it revokes* r *from user* u*, this is indicated by* $-r(u)$*. (E.g., obligation* b_1 *revokes the role* r_1 *from the user* u_1*). In the case of non-administrative obligations, this part of the annotation is empty. The current user role assignment we consider for the example is* $\gamma.UA = \{\langle u_0, r_0\rangle \langle u_1, \varnothing\rangle \langle u_2, r_0\rangle \langle u_3, \varnothing\rangle\}$.

Definition 23 (Positive Dependency)**.** *Given a system state* $s = \langle U, O, t, \gamma, B\rangle$*, a set of policy rules* $\mathcal{P}$*, an obligation* $b \in B$ *and a set of pending obligations* $\hat{B} \subseteq B$*, such that* $(\forall \hat{b} \in \hat{B} \cdot \hat{b}.end < b.end)$*. We say* b *has a* positive dependence *on* $\hat{B}$*, denoted by* $\hat{B} \xrightarrow{+} b$*, if and only if removing* $\hat{B}$ *from* B *yields an obligation pool in which* b *is not guaranteed to be authorized during its entire time interval and* $\hat{B}$ *is a minimal set satisfying this property.*

The arrow direction signifies that $\hat{B}$ is establishing in the authorization state the necessary permissions required by b.

Example 24. *In figure 5.1 we have the following positive dependencies* $\{b_2\} \xrightarrow{+} b_4$*,* $\{b_3\} \xrightarrow{+} b_4$*,* $\{b_6\} \xrightarrow{+} b_8$*,* $\{b_7\} \xrightarrow{+} b_8$ *and* $\{b_{11}, b_{10}\} \xrightarrow{+} b_{12}$*. Here,* $\{b_2\} \xrightarrow{+} b_4$ *and* $\{b_3\} \xrightarrow{+} b_4$*, because without* b_2 *and* b_3*,* b_4 *cannot be performed. The same is true in the case of obligation* b_8*. Note that* b_{12} *will be authorized if one of* b_{10} *and* b_{11} *is absent, but when both of them are absent,* b_{12} *will be not be authorized. Thus,* b_{12} *does not have a positive dependence on* b_{10} *or* b_{11} *individually, but* does *have a positive dependence on the set* $\{b_{11}, b_{10}\}$.

Definition 25 (Negative Dependency)**.** *Assume we are given a system state* $s = \langle U, O, t, \gamma, B\rangle$*, a set of policy rules* $\mathcal{P}$*, an obligation* $b \in B$ *and a set of pending obligations* $\hat{B} \subseteq B$*. We say that* b has a negative dependence on $\hat{B}$ *if* $\hat{B}$ *is a minimal set satisfying the following property. The start time of* b *is before that of each element of* $\hat{B}$ $($*i.e.*$, \forall \hat{b} \in \hat{B} \cdot \hat{b}.start > b.start)$ *and if* b *is rescheduled so that it starts after obligation in* $\hat{B}$*, then* b *is no longer guaranteed to be authorized throughout its entire time interval. In this case we write* $b \xleftarrow{-} \hat{B}$.

The direction of the arrow indicates that $\hat{B}$ yields an authorization state in which b may not be authorized during its entire time interval.

Example 26. *In figure 5.1, we have $b_4 \overset{-}{\longleftarrow} \{b_5\}$ due to the fact that if we reschedule b_4 after b_5, it will not be authorized.*

Definition 27 (Antagonistic Dependency)**.** *Given a system state $s = \langle U, O, t, \gamma, B \rangle$, a set of policy rules $\mathcal{P}$, and three obligations $b_1, b_2, b_3 \in B$, we say that b_2 has an* antagonistic dependence *on b_1 via b_3, denoted by $b_2 \overset{b_3}{—} b_1$, if inverting the order of b_1 and b_2 may result in there being a point during the interval of b_3 at which b_3 is not authorized.*

Example 28. *In figure 5.1, we have the following antagonistic dependencies $(b_1 \overset{b_4}{—} b_2)$, $(b_6 \overset{b_8}{—} b_9)$ and $(b_7 \overset{b_8}{—} b_9)$. Note that, b_7 and b_9 have an antagonistic dependency via b_8 although they consider different roles (viz., r_2 and r_1, respectively).*

5.2 Slice Properties

Having defined the dependence relations of interest, we now consider how to aggregate the dependencies at a higher level where they are more easily applied by the obligation system manager for the purpose to restoring accountability. The aggregation is a structure we call an *obligation-pool slice*, or simply *slice*. A slice is a subset $\hat{B}$ of a given obligation pool B. Intuitively, $\hat{B}$ consists of obligations that interact directly or indirectly with an input set of obligations $B_0 \subseteq B$ via various dependence relations relevant to the authorization requirements of obligations. The slice satisfies $B_0 \subseteq \hat{B} \subseteq B$ and is given by the closure of B_0 under some operation defined in terms of the dependence relation. The formal definition of each specific slice depends on the nature of the dependence relation used in its construction. We next provide these formal definitions, along with theorems that characterize them in terms of accountability, as needed for their use in accountability restoration.

Definition 29 (Positive Dependency Slice)**.** *Assume we are given a system state $s = \langle U, O, t, \gamma, B \rangle$, a set of policy rules $\mathcal{P}$, and a set of pending obligations $B_0 \subseteq B$. $PS_B(B_0)$ is the* positive dependency slice of B with respect to B_0 *if it is given by* $PS_B(B_0) = B_p$ *in which* $B_p \subseteq B$ *is the smallest set that satisfies the following requirements:*

- $B_0 \subseteq B_p$
- $\forall b \in B \cdot (\exists \check{B} \cdot (\check{B} \subseteq B_p \wedge \check{B} \xrightarrow{+} b) \longrightarrow (b \in B_p))$

The relation $\xrightarrow{+}$ is of type $2^B \times B$ where B is the current pending pool of obligations. The intuition behind calculating the positive dependency slice is to start the slice to be the set of obligations B_{v}. Then, we consider all the subsets of the current slice and check whether there is any obligation that is not part of the current slice that has a positive dependency on one of those subset. If we can find such an obligation we add it to the current slice. We continue this process until the size of the slice does not grow anymore.

Please note that the above procedure is provided here solely to assist the reader's intuition regarding the slice definition. The design of an efficient algorithm remains an open problem. At minimum, it should compute the dependence relation in a lazy way. Moreover, minimality in the dependence relation will be less of an issue in a real algorithm, which will focus on constructing the slice, not on the dependence relation that defines it.

Note that this notion of slice would be useful when the administrator is considering removal of pending obligations as (part of) her strategy for restoring accountability. The following theorem shows the utility of this slice with respect the to administrators objective of leaving the obligation pool in an accountable state.

Theorem 30. *Given an accountable system state* $s_0 = \langle U, O, t, \gamma, B \rangle$, *a policy* $\mathcal{P}$, *a set of obligations* $B_0 \subseteq B$, *and* $B_p = PS_B(B_0)$, *the state given by* $s = \langle U, O, t, \gamma, B \setminus B_p \rangle$ *is accountable.*

Proof. Suppose for contradiction that s is not accountable. In this case, there must be an obligations $\tilde{b} \in (B \setminus B_p)$ that is not guaranteed by $B \setminus B_p$ and γ to be authorized during its entire time interval. By the assumption that s_0 is accountable, either (1) $\tilde{b}$ is authorized during its entire time interval by γ and no obligations in B modified that part of γ on which $\tilde{b}$'s authorization depends, or (2) some of the obligations in B modified the authorization state so as to make $\tilde{b}$ authorized throughout its time interval. In case (1), it is impossible to remove obligations from B with the result that $\tilde{b}$ becomes possibly unauthorized, contradicting the assumption that s is not accountable. In case (2), B_p has the property that removing it from B yields an obligation pool in which $\tilde{b}$ is not guaranteed to be authorized during its entire time interval. It follows that B_p has at least one minimal subset satisfying this property. Call it $\hat{B}$. By definition of positive dependency, $\tilde{b}$ has a positive dependence on $\hat{B}$. Now by definition of positive dependency slice, it follows that $\tilde{b} \in B_p$, giving us the desired contradiction with $\tilde{b} \in (B \setminus B_p)$. □

Definition 31 (Full Dependency Slice)**.** *Assume we are given a system state $s = \langle U, O, t, \gamma, B \rangle$, a set of policy rules $\mathcal{P}$, and a set of obligations $B_0 \subseteq B$. The* full dependency slice *of B with respect to B_0, denoted by $FDS_B(B_0)$ is given by the smallest set B_f that satisfies the following properties:*

- $B_0 \subseteq B_f$
- $\forall b \in B \cdot (\exists \check{B} \cdot (\check{B} \subseteq B_f \wedge ((\check{B} \overset{+}{\longrightarrow} b) \vee (b \overset{-}{\longleftarrow} \check{B})))$
  $\longrightarrow (b \in B_f))$
- $\forall b \in B \cdot (\exists b_1 \in B_f \cdot \exists b_2 \in B \cdot ((b \overset{b_2}{—} b_1) \vee b_1 \overset{b}{—} b_2)$
  $\longrightarrow (b \in B_f))$

When the administrator is considering rescheduling the violated obligations and all its dependent obligations, this notion of slice would be used.

Theorem 32. *Given an accountable system state $s_0 = \langle U, O, t, \gamma, B \rangle$, a policy $\mathcal{P}$, a set of obligations $B_0 \subseteq B$, and $B_f = FDS_B(B_0)$, the state given by $s = \langle U, O, t, \gamma, B \setminus B_f \rangle$ is accountable.*

Proof. This theorem can be proved in a manner similar to that used in the proof of theorem 30. □

5.3 Restoring Accountability

In this section, we present several possible techniques by which an administrator can restore accountability. The selection among the techniques is application and system-requirement dependent. In practice, the administrator will use a combination of these techniques. Some obligations will, of course, be too important just to drop. Among these may be user-level obligations that do not change the authorization state. Achieving the intended changes to the authorization state might also influence the administrator's decision whether to drop obligations (including the violated ones), or instead to reschedule or reassign them.

It is important to bear in mind that restoring accountability while preserving all the desired obligations is not always possible. For instance, if an obligation with a hard deadline has been violated, this situation cannot be reversed. Furthermore, even when a solution exists, enabling us to reorganize existing obligations and add new obligations with the result that all desired obligations are fulfilled, it is not always going to be possible to find that solution in practice, as the problem is fundamentally intractable. Thus the support techniques and tools we discuss in section can at best increase the likelihood of finding a satisfactory solution. In the following we take B_0 to be the set of obligations that either have been violated or are unavailable.

Removal of Obligations When applying the *removal strategy*, the user removes the entire positive slice $B_p = PS_B(B_0)$ from the obligation pool B. The resulting obligation pool is accountable, as shown by theorem 30. Among the strategies for restoring accountability, this one modifies the fewest obligations, owing to the minimality of the sets in the forward dependency relation. Of course it may often be undesirable, depending on the importance of some of the obligations in the positive slice. However, sometimes there is really no alternative, since some deadlines are hard.

Example 33 (Removal of Obligations)**.** *Using the example in figure 5.1, consider b_2 and b_{11} have been violated. If we use the removal strategy, we need to remove obligations b_2, b_{11}, and b_4. We have to remove b_4 as it has a positive dependency on b_2.*

Rescheduling of Obligations In this approach, we can take advantage of the fact that some pairs of obligations in $B_f = FDS_B(B_0)$ are independent. In particular, B_f can be partitioned into sets such that obligations from different sets are independent on one another. of obligations where obligations in different partitions are independent of each other. In this case, each partition can be rescheduled independently of one another. We denote each partition of B_f as $B_f^i \subseteq B_f, 0 \leq i \leq |B_0|$.

Example 34 (Partitions of B_f)**.** *Using the example in figure 5.1, let the current system time be 15, and that* b_4 *and* b_{11} *have been violated. Thus,* $B_f = \{b_4, b_{11}, b_5\}$ *creates two partitions* $B_f^1 = \{b_4, b_5\}$ *and* $B_f^2 = \{b_{11}\}$.

For each B_f^i, we find the set of obligations $B_0^i \subseteq B_f^i$ that have already been violated or are unavailable. We assume an obligation $\tilde{b}$ has already been violated if $\tilde{b}.\mathrm{start} \leq t_c$ where t_c is the current system time. We use t_s and t_e to denote the earliest start time and the latest end time among all the obligations in B_0^i. Next, we find the obligation b_n with the earliest start time among all the obligations in $B_f^i \setminus B_0^i$. We then check whether it is possible to reschedule the obligations in B_0^i after t_c but before $b_n.\mathrm{start}$ $((t_e - t_s) < (b_n.\mathrm{start} - t_c))$. If so, we reschedule all the obligations in B_0^i after t_c keeping their original relative distance. If this is not the case, then we add b_n to the set B_0^i, and repeat the steps presented above (*i.e.*, compute t_e, and find a new b_n), until we find a time interval that is large enough to fit all the obligations in the current B_0^i. If no such intervals are found, we shift all the obligations in B_f^i so that the obligation with the earliest start time is scheduled at time $t_c + 1$ and the obligations maintain their original relative positioning. The intuition behind this approach is that all the obligations that can interact with each other will maintain their original relative positions and will be authorized.

Example 35 (Rescheduling of Obligations)**.** *the current time be* 8 *and* b_2 *and* b_3 *have been violated. If we reschedule* b_2 *and* b_3*, we need to reschedule the entire set* $B_f = \{b_2, b_3, b_4, b_5\}$*. The new time windows for the set could be* $b_2 = [21, 25]$*,* $b_3 = [22, 26]$*,* $b_4 = [28, 33]$ *and* $b_5 = [35, 38]$.

In some cases, it might not be possible to use the above approach. For instance, it may be essential that one of the obligations not be delayed. In such cases, the administrator must keep the time window of this obligation fixed, and attempt reschedule the other dependent obligations around it. However, if this is not possible, then the administrator may consider shrinking the width of some of the dependent obligations' time windows. Note that, every time the administrator tries to shrink the time window of an obligation, the reference monitor needs to check if the system is still accountable. It is up to the administrator to decide which obligations' time windows can be shrunk and by how much.

Reassignment of Obligations In this strategy, the administrator reassigns new users to the obligations that are unavailable. When an obligation's window has already passed, this approach must be combined with rescheduling. This case is discussed below under "hybrid strategy." Along with the reassigning technique, the administrator may use an AI-planner [47] to check what other actions (*e.g.*, giving required permissions to the new users) are required in order to transfer these obligations to the new users.

Example 36 (Reassignment of Obligations)**.** *Returning again to the example in figure 5.1, suppose the current system time be 4 and that the user u_0 will be unavailable within the time window [10, 13]. The administrator can reassign obligations b_{10} to user u_2, since u_2 has the role r_0 required for performing b_{10}.*

Addition of Obligations In this strategy the administrator adds new obligations in order to make the system accountable. (*E.g.*, let us consider obligation b_x needs a permission given by b_y which is scheduled before b_x. If b_y is violated, then the administrator can restore accountability by adding an obligation before b_x that grants the necessary permissions to it.) Again, the administrator can utilize the AI-planner presented in [47] to identify the new obligations she needs to add to the obligation pool to restore accountability when it has been violated.

Hybrid Strategy Obligations can be deemed by the administrator to have different levels of importance. It may be reasonable to remove some obligations, while other must be performed according to their original schedule. Moreover, some obligations that have been violated may have had hard deadlines and cannot be rescheduled or reassigned. Thus, the administrator requires the flexibility to apply a mixture of strategies to restore accountability. We propose a *hybrid strategy* for this purpose. In it, the administrator takes an incremental approach to constructing a solution to the accountability violation. The techniques presented above are applied to different violated or unavailable obligations and different portions of their slices.

Suppose the administrator decides to divide B_0 into three subsets: obligations that must be removed (B_0^{Rem}); obligations that must be reassigned (B_0^{Rea}); and obligations that must be rescheduled (B_0^{Res}). Of course, these choices cannot be made independently of one another. For instance, it is not possible to reschedule an obligation that depends on an obligation that will be removed. On the other hand, some obligations can be both rescheduled and reassigned.

The administrator then computes the positive dependency slice of B_0^{Rem}, denoted by B_p^{Rem}, and the full dependency slice for B_0^{Res}, denoted by B_f^{Res}. As discussed in "removal of obligations", if the administrator needs to remove the obligations in B_0^{Rem}, she also has to remove the obligations in B_p^{Rem}. Moreover, for rescheduling obligations in B_0^{Res}, she also has to reschedule obligations in B_f^{Res}. If B_p^{Rem} and B_f^{Res} intersect then removing B_p^{Rem} could yield an authorization state where rescheduling obligations in $B_f^{\text{Res}} \setminus B_p^{\text{Rem}}$ would not yield an accountable system. When B_p^{Rem} either contains any non-administrative obligation that is important or contains any administrative obligation that yields an authorization state necessary for discretionary actions, then she can not also remove the set B_p^{Rem} to yield an accountable system. In such cases, she tries to find a maximal subset of B_0^{Rem}, denoted by $\hat{B}_0^{\text{Rem}}$, so that the positive slice of it has neither any intersection with the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ nor does it contain any obligations that the administrator is unwilling to remove.

If she can find such a maximal subset, she can remove the obligations in the positive slice of of $\hat{B}_0^{\text{Rem}}$. Then, she can reschedule the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$. Finally, for reassigning the obligations in B_0^{Rea} she can use the approach discussed in "reassignment of obligations".

Rescheduling the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ can also introduce incompatibility. For instance, the administrator might not want to reschedule some obligations in the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ because of their urgency. We can address this problem in a manner similar to that discussed under "rescheduling of obligations".

Each time an administrator uses one of the techniques presented above to restore accountability, it is the system's responsibility to ensure that it is still in an accountable state. This can be checked using the algorithm in chapter 3.

5.4 Summary

In this chapter, we have introduced three different notions of authorization dependency among obligations. We have also presented formal specifications of two different notions of slice that calculates the set of obligations, which the administrator needs to consider when applying the different restoration techniques given a set of violated obligations. Finally, we have shown also a set of strategies that an administrator can use to restore accountability.

CHAPTER 6: CASCADING OBLIGATIONS

In chapter 2, we introduce policy rules for a user obligation system and they have the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. Such a policy rule p permits a user u to take an action a on the tuple of objects $\vec{o}$ if the user u fulfills the condition $cond$ (which is typically the authorization requirement of the system). The other component in the policy rule is $F_{obl}(s, u, \vec{o})$. This function returns a possible empty set of obligations that u or some other user in the system will incur when u executes the action a. The stipulated time window of the newly incurred obligations depend on the time when the action a is executed. Suppose the action currently being performed is an obligatory action. Although this possibility is not treated by our prior chapters, it could in general incur additional obligations depending on $F_{obl}(s, u, \vec{o})$. Those obligations could in turn incur further obligations, resulting in a chain of obligations, the time windows of which depend on each other, and on when, within its time interval, each obligation is performed. We call this phenomenon of obligations incurring additional new obligations, *cascading obligations*.

While our abstract model supports arbitrary cascading of obligations, the techniques discussed in chapter 2 for our concrete model disallow them. This is done by partitioning the actions into two disjoint groups, namely, discretionary action group and obligatory action group. We restrict the policy rules such that only actions from the discretionary group can introduce new obligations in the system. Thus, actions from obligatory action group can not further introduce new obligations. We impose this restriction due to several technical challenges, discussed just below, that we face when modeling arbitrary cascading of obligations in our model while attempting to maintain accountability.

It is our vision that interesting applications, such as project management tools and organizational workflows, can utilize user obligation systems for managing obligations. In such environments, cascading obligations appear naturally, due to the inherent capability of cascading obligations to capture dependence among various actions, obligatory or otherwise. For example, cascading obligations can be used to encode an organization's sales workflow: when a sales assistant submits a purchase order, the company clerk is obligated to issue a check in the amount identified in the purchase order. As soon as the clerk issues the check, the manager incurs an obligation to verify the consistency of the purchase order and approve the check. Thus, so it important to accommodate cascading in some manner.

We believe that workflows can be supported within our system if the form of policy rules we support are generalized with respect to their ability to constrain applicability of a given policy rule. Currently, only the authorizations of the user are checked. By constraining other aspects of the action parameters (such as the objects to which the action is applied) we will be able to encode decision points of workflows. However, we must extend our concrete model to allow cascading obligations.

It must be acknowledged that a great deal of work has been done in the area of workflow systems (see chapter 7). With few exceptions, workflow systems do not permit modification of authorizations. Moreover, those workflow systems that do permit modification of authorizations are unable to discover that a user lacks authorizations to perform an assigned task prior to the time at which that task is attempted. By maintaining accountability, we enable such errors to be detected and resolved much earlier. This is the main distinguishing feature of modeling workflows within the context of our authorization-aware obligation systems.

Some of the technical challenges to supporting cascading obligations while maintaining accountability are as follows:

1. Different policy rules that permit the same action can cause different obligations to be incurred. This makes it difficult at accountability-determination time to reason about the future state of the obligation pool. Certainly, if one rule leads to an accountable obligation pool, and the other does not, one would expect the former rule to be used. When both rules lead to accountability, the appropriate course of action seems to be application dependent. In some cases, it may be appropriate to let the user requesting the action make the decision. But this might be inappropriate for some situation, for example, systems that are concerned about performance (*e.g.*, response time, throughput rate).

2. The time intervals of the new obligations depend on the time at which the action is performed that causes them to be incurred. Making it difficult to reason about the appropriate time window when the new obligations must be authorized. Figure 6.1 presents two obligations, namely, b_1 and b_2. Let us assume that b_2 is incurred when b_1 was executed, in time t_x. Thus, b_2's time window will be given by a predefined distance from t_x. This is described in the picture as σ_x. Thus, depending on when b_1 is performed, b_2 will have different time windows. And it will be necessary to check accountability for each one of these time windows.

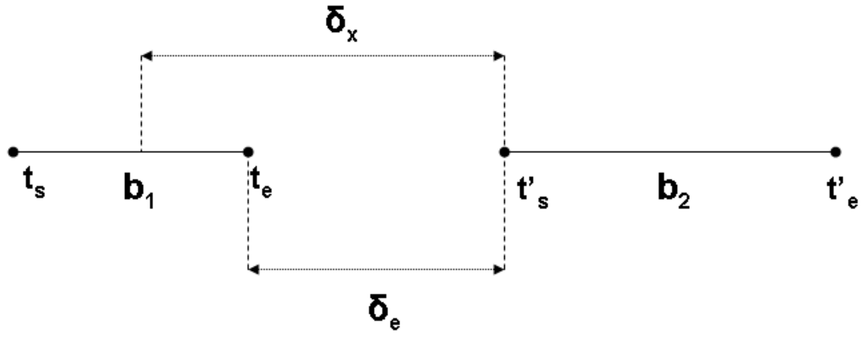

Figure 6.1: Formulation of Cascading Obligations

3. It is not trivial how to compute accountability when the policy allows infinite cascading. As we have finite number of users, objects, actions and policy rules in our system, to have an infinite chain of cascading obligations we must have cycles in the policy. If such a cycle has a regular behavior it is possible to have an efficient strong accountability decision procedure. But there are situations where this is not the case. One such critical situation appears when an obligation is authorized by multiple policy rules and each policy rule generates a different set of obligations. The same situation continues for the newly incurred obligations resulting an exponential increase of the number of obligations that one need to reason about for determination of accountability. Moreover, each such branch according to the choice of policy rule can have disjoint cycles of cascading obligations making the analysis difficult. Thus, if the number of obligations in the current pool is high and they are well spread in a large time window, such a situation discussed above can make the strong accountability determination

difficult.

6.1 Strong Accountability under Cascading of Obligations

In this section, we provide a definition of strongly accountability under the cascading of obligation assumption. We start by defining two auxiliary functions that are going to be used in the definition of strong accountability.

Definition 37 (F_{obls}). *F_{obls} is a function that takes as input an obligation $\hat{b}$ and returns a set of sets of obligations $\hat{B}$ in which each element represents a set of obligations that $\hat{b}$ can incur according to the F_{obl} function of a policy authorizing it. The formal specification and the type of F_{obls} are precisely shown just below.*

$$F_{obls} : B \rightarrow \mathcal{FP}(\mathcal{FP}(B))$$
$$F_{obls}(b = (u, a, \vec{o}, t_s, t_e)) = \{F_{obl}(u, \vec{o}) | p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o}) \in \mathcal{P}\}$$

Definition 38 (*PF*$_{obls}$). *PF$_{obls}$ is a function that takes as input a set of obligations $\bar{B}$ and returns a set of sets of obligations $\tilde{B}$ in which each element is a possible set of obligations that all the obligations of $\bar{B}$ can incur. In short, $\tilde{B}$ is the set containing all possible combination of obligations that $\bar{B}$ can incur. The formal specification and the of type of the function PF$_{obls}$ are shown just below.*

$$PF_{obls} : \mathcal{FP}(B) \rightarrow \mathcal{FP}(\mathcal{FP}(B))$$
$$PF_{obls}(b_{1...n} = (u_{1...n}, a_{1...n}, \vec{o}_{1...n}, t_s, t_e)) = \{BB \subseteq \mathcal{B} | \forall i \in 1 \ldots n. F_{obls}(b_i) \neq \emptyset \rightarrow \exists f \in F_{obls}(b_i). f \subseteq BB\}$$

Definition 39 (Strong accountability in presence of Cascading of Obligations). *Given a state $s_1 \in \mathcal{S}$, in which $s_1.B$ is a strongly accountable pool of obligations, a policy $\mathcal{P}$, a set of new obligations B_c that can generate cascading of obligations, we say that the state s created by the union of $s_1.B$ and B_c is strongly accountable (denoted by $sac(s_1, B_c, \mathcal{P})$) if and only if $sa(s, \mathcal{P}) \wedge s =$*

$s_1[B := s_1.B \cup B_c] \wedge \exists B'_c \in \mathcal{P}.\boldsymbol{PF}_{obls}(B_c).sac(s, B'_c, \mathcal{P})$. *Where* $sa(s, \mathcal{P})$ *is the definition of strong accountability without cascading (see definition 11 presented in chapter 3).*

6.2 Cascading of Obligations

Our obligation system is governed by policy rules that have the following format, $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. If a user u satisfies the authorization requirement defined by $cond$, he can take the action associated by the policy rule. When the user takes the action, the policy rule may in turn create more obligations. This is governed by the $F_{obl}(s, u, \vec{o})$ of the policy rule. In our concrete model, we did not define how F_{obl} generates the parameters of newly incurred obligations. In this section, we present options which can be used to instantiate the parameters of the newly incurred obligations. For this, we explain how the policy rules encode the new obligations' time windows, how users are selected, *etc.* Then, we present a taxonomy that describes the types of cascading obligations that our obligation model supports.

Number of Policy Rules Obligations are incurred by policy rules. Depending how the policy rule is written we can have:

1. **Obligation authorized by multiple policy rules:** Here, an obligation can be authorized by many policy rules. Different policy rules that permit the same obligation can cause different obligations to be incurred.

2. **Obligation authorized by only one policy rule:** Here, an obligation is authorized by only one policy rule. Thus, the cascading of obligations that will be incurred by an action object pair will always be the same.

Selection of Users When someone executes an action, this can generate other obligations to the user that initiated the action, or for other users. The user who incurs the obligations is called obligate. To allow to specify the obligatee we should extend our policy rules to include the obligatee. Below, we present how the selection of users can be incorporated to the policy rules:

1. **Explicit User:** In this strategy, the obligatee is hard-coded in the policy rule.

   Example 40 (Selection of Explicit User)**.** *Let us assume a permission assignment rule,* p_1 = $\{\, manager,\ check,\ log\}$: F_{obl} = $\{Bob,\ send,\ report,\ shift = 10,\ window = 5\}$*. This rule tells that a user is going to be authorized to check the log if she is in the role of manager. Once, she checks the log, this will incur an obligation to Bob to send the report.*

2. **Self, Target, and Explicit User :** When a policy rule's obligatee field contains "Self", it precisely specifies that the user who initiates the action authorized by the policy will get the obligation incured by it.

   Example 41 (Self clause)**.** *Let us assume a can assign rule,* p_1 = $\{\, manager,\ employee,\ programmer\}$: F_{obl} = $\{Self,\ send,\ report,\ shift = 10,\ window = 5\}$*. This rule tells that a user in the role of manager is authorized to grant the role programmer to a user in the role of employee. Let us assume the manager Bob grants the employee Alice to the role of programmer. This will generate a new obligation to Bob (as he is the originator of the obligation, viz. Self) to send a report.*

   On the other hand, whenever the policy rule is authorizing an administrative action and the obligatee field of that policy rule contains "Target" as an argument, it signifies that the target of the original administrative action authorized by this policy would incur the obligations specified by it.

   Example 42 (Target clause)**.** *Let us assume a can assign rule,* p_1 = $\{\, manager,\ employee,\ programmer\}$: F_{obl} = $\{Target,\ send,\ report,\ shift = 10,\ window = 5\}$*. This rule tells that a user in the role of manager is authorized to grant the role programmer to a user in the role of employee. Let us assume the manager Bob grants the employee Alice to the role of programmer. This will generate a new obligation to Alice (as she is the target of the original obligation, viz. target) to send a report.*

   Note that, the obligatee may also be hard-coded in the policy rule.

3. **Role expression :** In this approach the obligatee contain boolean role expression. Each literal in the boolean expression can contain a positive or negative role assignment. The system can select a user to be the obligatee provided the user satisfies the role expression when the new obligation is incurred.

   Example 43 (Role expression)**.** *Let us assume a permission assignment rule,* $p_1 = \{\ manager,\ check,\ log\} : F_{obl} = \{intern \wedge \neg programmer,\ send,\ report,\ shift = 10,\ window = 5\}$. *This rule tells that a user is going to be authorized to check the log if she is in the role of manager. Once, she checks the log, the system will randomly select a user that is in the role of intern and but is not a programmer, and it will create an obligation to this user to send the report.*

Creation Time Recall from the previous sections, that the time window of the new obligation is a constant displacement from the action/obligation that incurs the new obligation. Thus, it is necessary to add two new fields in the F_{obl}, namely, the shift, and window size. The shift determines the distance between the new obligation, and the obligation that incurs it. Whereas the window size tells the size of the time window of the new obligation. Depending on how the time is shifted from the original obligation, the policy rule can follow two different strategies. We consider them below:

1. **Based on the execution time :** In this strategy, the time window of the new obligation will depend on the execution time of the action/obligation that incurs the new obligation. For instance, if obligation b_1 generates obligation b_2, and the policy rule that governs b_1 specifies that the shift is 10 and the new obligation window size is 5. Then, if b_1 is executed in time 6, then the time windows for obligation b_2 will be $b_2.start$ = 16 and $b_2.end$ = 21.

2. **Based on obligation time window :** Here, the time window will depend on the end time of the action/obligation that is incurs the new obligation. For instance, if obligation b_1 generates obligation b_2 and the time window of b_1 is $[2, 7]$. The policy rule that governs b_1 specifies

that the shift is 10 and the new obligation window size is 5. Then, no matter when b_1 is executed in its time window, the time window for obligation b_2 will be $[17, 22]$.

6.2.1 Taxonomy

In this section, we present the types of cascading obligations that can be supported by our concrete model.

Repetitive obligation as its name suggests are obligations that occur recurrently after a fixed amount of time (*e.g.*, A security officer needs to investigate the system logs every week). In order to represent repetitive obligations, we modify the notion of temporal constraints as presented by Ni *et al.* [41] by adding an additional field named "shift". The shift signifies the distance between two occurrence of repetitive obligations.

Definition 44 (Temporal Constraint)**.** *A temporal constraint is a tuple* $(\langle t_s, t_e\rangle, shift, repetition)$*, where* $t_s, t_e, shift \in \mathbb{N}$*, and* $repetition \in \mathbb{N}*$ *or* $repetition = I$*.*

A temporal constraint represents a sequence of time intervals defined as follows:

- $[t_s, t_e], [t_s + shift, 2t_e + shift], ...[t_s + (repetition - 1)(t_e - t_s + shift), t_e + (repetition - 1)(t_e - t_s + shift)]$. If $repetition \in \mathbb{N}^*$
- $[t_s, t_e], [t_s + shift, 2t_e + shift], ...[t_s + (repetition - 1)(t_e - t_s + shift), t_e + (repetition - 1)(t_e - t_s + shift)], ...$ If $repetition = I$

We augment our obligation model to support repetitive obligations. Now, obligations have temporal constraints instead of time windows.

- **Finite Repetitive Obligations** occurs recurrently after a fixed amount of time for a finite amount time. It is defined as $b = \{user,\ action,\ objects,\ \langle t_s, t_e\rangle,\ shift,\ repetition\}$. Where $repetition \in \mathbb{N}^*$. For instance, $b = \{Bob,\ check,\ log,\ \langle 5, 8\rangle,\ 2,\ 3\}$ will generate 3 obligations $\{Bob,\ check,\ log,\ \langle 5, 8\}, \{Bob,\ check,\ log,\ \langle 10, 13\}, \{Bob,\ check,\ log,\ \langle 15, 18\}$.

- **Infinite Repetitive Obligations** occurs recurrently after a fixed amount of time. It is defined as $b = \{user,\ action,\ objects,\ \langle t_s, t_e \rangle,\ shift,\ repetition\}$. Where $repetition = I$. For example, $b = \{Bob,\ check,\ log,\ \langle 5,8 \rangle,\ 2,\ I\}$ will generate infinite obligations, $\{Bob,\ check,\ log,\ \langle 5,8\},\{Bob,\ check,\ log,\ \langle 10,13\},...$

Directly cascading of obligations describes a cascading of obligations in which each is authorized by only one policy rule. In addition, the selection of users will use the "Self, Target, and Explicit user" strategy. The time window of the new obligations will depend on the end time period of the action/obligation that incurs the new obligations. The policy rules are guaranteed to be free of cycles. In this way, an obligation can only generate a finite set of obligations. These obligations do not incur repetitive obligations.

Unrestricted cascading of obligations describes unrestricted cascading of obligations. Here, obligations can be authorized by multiple policy rules. The time window of the new obligations will depend on the execution time of the action/obligation that originated them. Policy rules can have cycles. In this way, that an obligation can generate an infinite set of obligations. In addition, the selection of users combines all the strategies presented before.

6.3 Proof of co-NP completeness of Strongly Accountability Problem

Theorem 45. *Given a strongly accountable pool of obligations B, a new obligation b, an initial authorization state γ, and a mini-ARBAC policy ψ that allows cascading of obligations and also allows each action to be authorized by multiple policy rules, deciding whether $B \cup B_c$ is strongly accountable is NP-hard in the size of B, γ, and ψ, where B_c is the set of cascading obligations incurred by b.*

Proof. To show that deciding accountability is NP-hard when the input mini-ARBAC policy allows cascading of obligations and also allows each action to be authorized by multiple policy rules, we reduce the *Hamiltonian path problem* for directed graphs to it. Given a directed graph $G(V, E)$

where V is the set of vertices and E is the set of edges, the Hamiltonian path problem asks whether there is a simple path in the graph $G(V, E)$, such that it contains all the vertices and each of the vertices in V are visited only once in that path. To this end, we present a polynomial time algorithm which reduces a Hamiltonian path problem instance to the problem instance of deciding accountability in presence of cascading obligations. Thus, the graph will have a Hamiltonian path when the reduced problem instance of deciding accountability[1] yields true.

Let us consider a Hamiltonian path problem instance where the directed graph $G(V, E)$ is given. Let us also assume that the total number of vertices in the graph is n (*i.e.*, $|V| = n$). We also consider the vertices in V are uniquely labeled using a number from $1...n$. Each edge $e \in E$ has the form (v_i, v_j) where $1 \leq v_i, v_j \leq n$.

Now, we will try to construct a problem instance of deciding accountability that corresponds to the Hamiltonian path problem instance. In the accountability decision problem instance, consider that we have only two users, namely u_0 and u_1. Furthermore, let us also consider we have the following roles, $ar_1, r_0, \hat{r}, r_{1...n}, v_{1...n}$. Each of the role $v_{1...n}$ corresponds to the vertex of the graph. At each point of time, the roles (in $v_{1...n}$) that the user u_1 currently possesses, denote the vertices that we have already visited in the current path in the graph. Additionally, each of the role $r_{1...n}$ also corresponds to the vertices of the graph. This is used to determine the current vertex being inspected and will be explained later.

The current role assignment of users in the system is as following.

$$user\_assignment\{\langle u_0, \{ar_1\}\rangle, \langle u_1, \{\varnothing\}\rangle\}$$

Now, we turn our attention to the input mini-ARBAC policy of the problem instance. The policy for the corresponding problem instance would be like following:

For each vertex i in the graph $G(V, E)$, we have a can_assign rule, each of which has the

[1]When we say deciding accountability, we actually mean deciding the stronger version of the accountability

following form where $1 \leq i \leq n$.

$$\begin{aligned} &can\_assign(ar_1, \textit{true}, r_0) : \\ &\qquad \{ \\ &\qquad \langle u_0, Grant, u_1, r_i, 1, 1\rangle, \\ &\qquad \langle u_0, Grant, u_1, \hat{r}, 5 * n, 10\rangle \\ &\qquad \} \end{aligned} \tag{6.1}$$

For each edge $e = (x, y) \in E$, we have one policy rule like the following,

$$\begin{aligned} &can\_assign(ar_1, \neg v_x \wedge \neg v_y \wedge \neg r_y, r_x) : \\ &\qquad \{ \\ &\qquad \langle u_0, Grant, u_1, v_x, 1, 1\rangle, \\ &\qquad \langle u_0, Grant, u_1, r_y, 1, 1\rangle \\ &\qquad \} \end{aligned} \tag{6.2}$$

We also have the following policy rules for each of the vertices. The policy specifies that if r_i is the last role among the roles $r_{1...n}$ that is being granted to u_1 then there is no need to incur any further obligations and it could be the end of the simple path where each vertex is representing granting of a role to u_1.

$$can\_assign(ar_1, \bigwedge_{(1 \leq j \leq n) \wedge (i \neq j)} r_j, r_i) : \{\varnothing\}, 1 \leq i \leq n \tag{6.3}$$

The following policy rule allows a user in role ar_1 to grant the user who has all the roles in $r_{1...n}$ to be assigned the role $\hat{r}$.

$$can\_assign(ar_1, \bigwedge_{1 \leq i \leq n} r_i, \hat{r}) : \{\varnothing\} \tag{6.4}$$

In the problem instance, consider $B = \varnothing$ and the obligation we want to add is $b = \langle u_0, Grant, u_1, r_0, [1,2] \rangle$. Furthermore, we consider that when an action is authorized by multiple policy rules, then we select the one to use non-deterministically. We also consider that adding an obligation b in the accountable pool of obligations B will not violate the accountability property as long as there is a cascading pool of obligations B_c incurred by b, in which all the obligations in $B \cup B_c$ are authorized. However, in our instance, as B is empty, we just need to check whether there exists a B_c in which all the obligations are authorized.

Now when we consider the new obligation $b = \langle u_0, Grant, u_1, r_0, [1,2] \rangle$, no matter what policy rules of the form (1) we use, it will incur an obligation $\hat{b} = \langle u_0, Grant, u_1, \hat{r}, [5 * n + 2, 5 * n + 12] \rangle$. Thus, one of the pre-condition of b not violating accountability is that the obligation $\hat{b}$ be authorized. We can see from the policy rule (4) that it is the only policy rule that can possibly authorize it. It however requires that the user u_1 possesses all the roles in $r_{1...n}$. Now u_1 to get all the roles the policy rules that can be used are either of form (1), (2), or (3). These policy rules make sure that the only way a user can get all the roles in $r_{1...n}$ if there is a simple path in the corresponding graph containing all the vertices only once. The policy rules of form (1) ensure that one can start looking for such a path in any of the vertices of the graph. The policy rules of form (2) on the other hand encodes the edge relationship and also impose a constraint that the only way to get a role r_i through an obligation incurred due to granting a predecessor role (predecessor vertex) and if she did not have the role before (not visiting a vertex twice). The policy rules of form (3) precisely specifies that whenever u_1 possesses the last role of the roles in $r_{1...n}$, there is no need to search anymore as we have already found a simple path containing all the roles. Thus, $G(V, E)$ will have a Hamiltonian Path if the accountability decision procedure yields true.

□

Scope Given the above theorem, we provide a decision procedure for deciding strong accountability in presence of finite and infinite repetitive obligations, and also directly cascading of obligations.

6.4 Algorithm for Determining Infinite Cascading

This section presents an algorithm that can statically verify the absence of cycles among policy rules. By guaranteeing that our policy rules do not contain cycles, we avoid the creation of infinite cascading of obligations.

Algorithm 9 works as following. It proceeds in a depth first search manner inspecting each possible action, object pair in the system. It considers that they are authorized and checks what obligations are incurred due to executing the action. It then calls the procedure *findCycles* to check whether we can reach an action, object pair we have already seen before. If that is the case, it guarantees that there is a cycle in the policy.

Algorithm 9 *InfiniteCascading*($\langle\gamma, \psi\rangle$)

```
Input: A policy $\langle\gamma, \psi\rangle$
Output: Returns **true** if $\langle\gamma, \psi\rangle$ allows infinite cascading obligations
1: map⟨pair⟨action, object⟩, boolean⟩ obligationSeen
2: obligationSeen.clear()
3: for each possible action, object pair $\langle a, o\rangle$ in the system do
4:     obligationSeen.insert($\langle\langle a, o\rangle$, true⟩)
5:     if findCycles($a, o$, obligationSeen) == true then
6:         return true
7:     obligationSeen.delete($\langle\langle a, o\rangle$, true⟩)
8: return false
```

6.5 Algorithm for Determining Strong Accountability

This section begins by presenting our algorithm for determining whether adding a set of cascading of obligations to a strongly accountable obligation pool preserves the accountability property. It then discusses the complexity of the algorithm. The algorithms presented here are specialized to mini-ARBAC and mini-ARBAC.

6.5.1 The Algorithm

The idea behind Algorithm 11 is that we are going to add three different types of cascading of obligations to our obligations pool, and then decide whether the final pool is accountable. To this

Algorithm 10 *findCycles*(*action* a, *object* o, *map obligationSeen*)

```
Input: An action $a$, object $o$, and a map data structure which represents the obligations/action, object pairs we have already seen.
Output: Returns true if $\langle a, o\rangle$ can generate an infinite cascading of obligations
1:  for each possible policy rule $p \in P$ do
2:    if $p.a = a \wedge p.\vec{o}[0] = o$ then
3:      obligations $B = p.Fobl()$
4:      for each obligation $b \in B$ do
5:        if obligationSeen.find($\langle b.a, b.\vec{o}[0]\rangle$) = true then
6:          return true /* cycle found */
7:        else
8:          obligationSeen.insert($\langle\langle b.a, b.\vec{o}[0]\rangle$, true$\rangle$)
9:          if findCycles($b.a, b.\vec{o}[0]$, obligationSeen) = true then
10:           return true /* cycle found */
11:         obligationSeen.delete($\langle\langle b.a, b.\vec{o}[0]\rangle$, true$\rangle$)
12: return false
```

end, we unroll the chain of cascading obligations incurred by obligation b (*i.e.*, B'). We also unroll the set of finite repetitive obligations incurred by B_r (*i.e.*, B''), and finally, we unroll the infinite repetitive obligations incurred by B_{r_i} (*viz.*, B''').

Then, we used the non-incremental version of the strongly accountability algorithm presented in chapter 3 for deciding whether the whole set $B_{final} := B \cup B' \cup B'' \cup B'''$ is strongly accountable. The Algorithm 11 achieves this by adding each administrative obligation to an empty modified interval search tree and then calling *Authorized* (see Algorithm 2 in chapter 3) for each obligation $b' \in B_{final}$ to see whether it is authorized in the context of $\langle\gamma, \psi\rangle$ and B_{final}.

Unrolling Directly Cascading To unroll the chain of cascading obligations incurred by b, Algorithm 11 uses procedure *UnrollCascading* described in Algorithm 12. This procedure is an adaptation of the breadth-first search (BFS) algorithm. Recall that, we disallow infinite directly cascading of obligations. Thus, the procedure *UnrollCascading* is guaranteed to terminate. This procedure uses the function $Fobl$, which takes an obligation b_c as its input. It returns a set of obligations that are incurred as result to taking the action associated with b_c. The parameters of the obligations will depend on the policy rule used to authorize b_c.

Algorithm 11 *StrongAccountableCascading* $(\gamma, \psi, B, b, B_r, B_{r_i})$

Input: A policy $\langle\gamma, \psi\rangle$, a strongly accountable obligation set B, and a new obligation b that generates cascading obligations, a set of finite repetitive obligations B_r and a set of infinite repetitive obligations B_{r_i} .

Output: returns **true** if addition of b, B_i and B_r to the system preserves strong accountability.

```
1:  $B' := UnrollCascading(\gamma, \psi, b)$;
2:  $B'' := UnrollFiniteRepetitive(\gamma, \psi, B_r)$;
3:  $m := MaxEndTime(B, B', B'')$;
4:  $B''' := UnrollInfiniteRepetitive(\gamma, \psi, B_{r_i}, m)$;
5:  $B_{final} := B \cup B' \cup B'' \cup B'''$;
6:  for each obligation $b^* \in B_{final}$ do
7:      if $b^*.a$ = grant or revoke then
8:          InsertIntoDataStructure($b^*$);
9:  for each obligation $b^* \in B_{final}$ do
10:     if Authorized ($\gamma, \psi$, $B_{final}$, $b^*$)= false then
11:         return false
12: return true
```

Algorithm 12 *UnrollCascading* (γ, ψ, b)

Input: A policy $\langle\gamma, \psi\rangle$ and a new obligation b.

Output: returns a set of cascading obligations B that is generated by b.

```
1:  $B = \varnothing$;
2:  $queue < obligation > q$;
3:  q.push(b);
4:  while $!q.empty()$ do
5:      $b = q.front()$;
6:      $B := B \cup \{b\}$;
7:      $q.pop()$;
8:      $B' := Fobl(b)$;
9:      for each obligation $b^* \in B'$ do
10:         $q.push(b^*)$;
11: return $B$
```

Unrolling Finite Repetitive Obligations To unroll the obligations incurred by the finite repetitive obligations B_r, Algorithm 11 uses procedure *UnrollFiniteRepetitive* described in Algorithm 13. Recall that, in repetitive obligations there are two extra fields, namely, the shift, and the number of repetition. Thus, for each obligation b_r in B_r, the procedure creates copies of b_r, varying only the time window; the exact number of copies will depend on the field "number of repetition" predefined in b_r. Below, we present an example to illustrate this process.

Example 46 (Unrolling Finite Repetitive Obligations)**.** *Given a finite repetitive obligation* b =

$\{u_1, check, logFile, \langle 2,4\rangle, shift = 1,\ repetitions = 3\}$*, the unrolling process will yield to the following 3 obligations,* $b_1 = \{u_1, check, logFile, \langle 2,4\rangle\}$*,* $b_2 = \{u_1, check, logFile, \langle 5,7\rangle\}$ *and* $b_3 = \{u_1, check, logFile, \langle 8,10\rangle\}$.

Algorithm 13 *UnrollFiniteRepetitive* (γ, ψ, B)

```
Input: A policy $\langle\gamma,\psi\rangle$ and a set of finite repetitive obligations $B$.
Output: returns a set of unrolling obligations $B'$ that is generated by $B$.
 1: for each obligation $b' \in B$ do
 2:     $B' = \varnothing$;
 3:     $i := 1$;
 4:     $window := b'.t_e - b'.t_s$;
 5:     while $i \le b'.repetition$ do
 6:         $b_i := b'$;
 7:         $b_i.t_e := (b'.t_e - b'.shift - b'.t_s) * i + b'.t_s - b'.shift$;
 8:         $b_i.t_s := b_i.t_e - window$;
 9:         $B' := B' \cup \{b_i\}$;
10:         $i++$;
11: return $B'$
```

Unrolling Infinite Repetitive Obligations Algorithm 11 uses procedure *UnrollInfiniteRepetitive* to unroll the obligations incurred by the infinite repetitive obligations B_{r_i}. First, we find the *overall period* at which the infinite repetitive obligations will start to repeat themselves. In figure 6.2 we have two infinite repetitive obligations, $b_1 = \{u_1, a, o, \langle 1,5\rangle, shift = 1, repetitions = I\}$ and $b_2 = \{u_1, a, o, \langle 1,10\rangle, shift = 1, repetitions = I\}$. It is clear, that that after time 11, we start to see a pattern formed by the obligations, this is the overall period. To compute the overall period, we use procedure LCM. The procedure is used to find the least common multiple of a set of periods. We compute each obligation period in the following way, given a infinite repetitive obligation b_i, which can be unrolled to $b_0, b_1, ..., b_i$. The time that will be used for the LCM procedure is given by $b_1.s - b_0.s$. In this figure, the periods that are used to the LCM are 5 and 10, which will generate a overall period of size 10.

Once the period is computed, we check whether the period is greater than the maximum time of the finite obligations. If this is the case, we just need to unroll the infinite repetitive obligations until two additional periods. Otherwise, we unroll the infinite obligations until the maximum time,

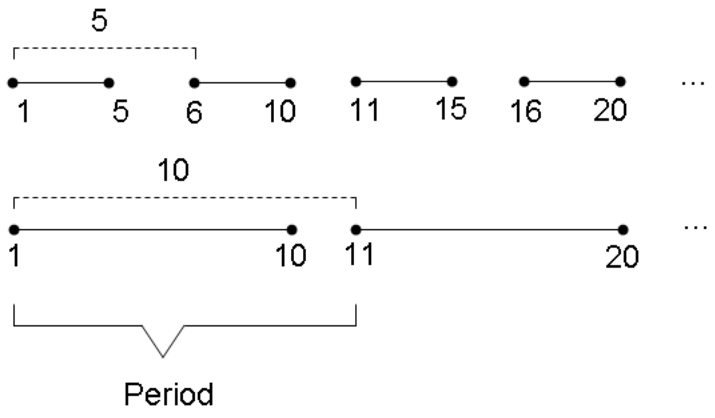

Figure 6.2: Computing the Least Common Multiple

and then we unroll one additional period. Figure 6.3 describe this procedure.

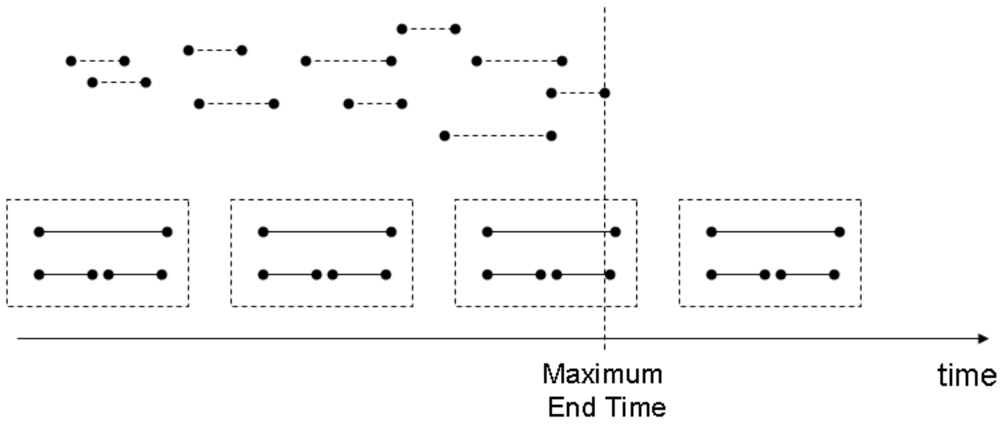

Figure 6.3: Unrolling Infinite Repetitive Obligations

Complexity Analysis Initially, we have an obligation set that contains $|B|$ obligations. After unrolling all the obligations, we end up with $n = |B| + |B'| + |B''| + |B'''|$. The cost to unroll the directly cascading of obligations is $\mathcal{O}(|B'|)$. The cost to unroll the finite repetitive obligations is $\mathcal{O}(|B''|)$. The cost to unroll the infinite repetitive obligations is $\mathcal{O}(|B_{r_i}|^2 + |B'''|)$.

So, the total cost is $\mathcal{O}(qmn^2 \log n + a \log n + |B'| + |B''| + |B_{r_i}|^2 + |B'''|) = \mathcal{O}(qmn^2 \log n)$, in which a is the number of obligations to perform administrative actions ($a \leq n$). (The term $a \log n$ is the cost of constructing the search tree.) q is the number of policy rules ($q = Max\{|CA|, |CR|, |PA|\}$) and m is the maximum size of the role constraints in the *can_assign* rules in CA.

Algorithm 14 *UnrollInfiniteRepetitive* (γ, ψ, B, m)

```
Input: A policy $\langle\gamma, \psi\rangle$, a set of infinite repetitive obligations $B$, and a point in time $m$ representing the last
    time point where a non-infinite obligation happens.
Output: returns a set of unrolling obligations $B'$ that is generated by $B$.
 1: $B' = \varnothing$;
 2: $period = LCM(B)$
 3: if $period > m$ then
 4:     $finalTime := period * 2$;
 5: else
 6:     $finalTime := (\lceil (m/period) \rceil + 1) * period$;
 7: for each obligation $b' \in B$ do
 8:     $end := b'.t_e$;
 9:     $window := b'.t_e - b'.t_s$;
10:     while $end <= finalTime$ do
11:         $b_i := b'$;
12:         $b_i.t_e := (b'.t_e - b'.shift - b'.t_s) * i + b'.t_s - b'.shift$;
13:         $b_i.t_s := b_i.t_e - window$;
14:         $B' := B' \cup \{b_i\}$;
15:         $end := b_i.t_e$;
16: return $B'$
```

Algorithm Variations One of the problems that Algorithm 11 can suffer is an explosion in the length of the chain of the cascading obligations. This can happen for three different causes. First, an obligation added to the system can trigger several other obligations. Although, this is possible, we do not consider this. As we are prohibiting infinite cycles in the policy rules, the amount of obligations triggered by one obligation would be bounded by the Fobls described in the policy rules. Secondly, one can add a finite repetitive obligation that needs to repeat itself several times. In this case, we can try to bound the number of times a finite repetitive obligation can happen. Finally, when someone adds infinite repetitive obligations, sometimes the period that these infinite repetitive obligations are going to repeat is very large. In this case, it is going to be necessary to unroll many infinite repetitive obligations. A possible solution for this problem is to impose a restriction in our system. Namely, finite and infinite repetitive obligations will be non-administrative in nature. That is, they will not alter the authorization state. In this case, we only need to unroll the obligations until after the maximum end time of non-repetitive obligations.

Algorithm 11 is receiving three different sets of obligations. Another possibility to the algorithm would be to receive only one obligation, that could be, a obligation that will incur a chain of cascading of obligations, or a finite repetitive obligation, or a infinite repetitive obligation. Currently, in the system there is a set of pending obligations, which finite repetitive obligations. And a separated set that contain not unrolled infinite repetitive obligations. Now, if someone wants to add an obligation that triggers other obligations, the algorithm unroll the chain of the obligations that will be incurred by this obligation and add to the pending obligation. Then, the algorithm recomputes the period of the infinite repetitive obligations and check whether the unrolled infinitive repetitive obligations together with the pending obligations are accountable. Note that, the algorithm will never mix the infinite repetitive obligations with the set of finite obligations. In the case that one wants to add an infinite repetitive obligation to the system. The algorithm adds this infinite repetitive obligation to the infinite repetitive pending obligation set, then it computes the new period, unroll the infinite repetitive obligations, and compute accountability to these obligations together with the finite pending obligation pool. The last case is analogous, but now if someone tries to add finite repetitive obligation. The algorithm unroll this finite repetitive obligation, and add this obligations to the finite pending obligation pool. Then, it recomputes the period for infinite repetitive obligations, it unrolls them, and it checks whether the union of both sets is strongly accountable.

6.6 Empirical Evaluations

The goal of these empirical evaluations is to determine the impact of restricted versions of cascading of obligations when deciding strong accountability when using our obligation model with mini-RBAC/mini-ARBAC as the authorization state.

Policy generation To evaluate the strong accountability algorithms for mini-ARBAC/mini-RBAC, we assumed 1000 users and used a handcrafted mini-ARBAC/mini-RBAC policies $\langle \gamma_0, \psi_0 \rangle$ summarized in table 3.2.

To evaluate the strong accountability algorithm under the cascading of obligations assumptions, we assume 1007 users and used a handcrafted mini-RBAC/mini-ARBAC policy that contains, 551 roles, 53 actions (2 administrative, 51 non-administrative), 1051 objects, 560 can assigns rules (with 5 maximum role preconditions), 560 can revoke rules, 1251 permission assignment rules. 101 Fobl rules, each Fobl generating 10 new obligations, making a total of 1000 new cascading of obligations generated.

To generate the obligations, we handcrafted 6 strongly accountable sets of obligations in which each set has 50 obligations. Each set has a different ratio of administrative to non-administrative obligations (rat). We then replicated each set of obligations for different users to obtain the desired number of obligations. Similarly, we generate the infinite and finite repetitive obligation in the same way, we have 6 sets of repetitive obligations that are strongly accountable. The execution times shown are the average of 100 runs of each experiment.

Experimental Environment All the strong accountability experiments are performed using an Intel i7 2.0GHz computer with 6GB of memory running Ubuntu 11.10.

6.6.1 Evaluations Results

Directly cascading For these empirical evaluations, we add one obligation to a strongly accountable obligation set. This obligation incurs 1000 new obligations. Then, the algorithm needs to answer whether these 1000 obligation along with the original strongly accountable obligation set is still strongly accountable. Figure 6.4 presents results for the strong accountability algorithm when considering the case described above. As we can see, the time required by the strong accountability algorithm grows roughly linearly in the number of obligations. In the worst case the algorithm runs in 110 milliseconds to determine that the set is strongly accountable. This is roughly two times slower than the non-incremental strong accountability algorithm presented in chapter 3 without cascading of obligations. This is due to the overhead of unfolding the cascading obligations (algorithm 13, which is a variation of the breadth first search). The impact of rat on the execution time of algorithm arises largely because the algorithm must inspect every obligation following each administrative obligation. When someone tries to add a unauthorized obligation to a strongly accountable set, the algorithm can determine that the final set is not accountable in less than 90 milliseconds as shown in figure 6.5. Which again is two times slower than the non-incremental algorithm presented chapter 3 without considering cascading of obligations.

Only finite repetitive obligations Here, we consider to add 50 finite repetitive obligations to a strongly accountable obligation set. These 50 obligations generate 1000 new obligations. The algorithm will check whether the old set of obligations plus this set is strongly accountable. Figure 6.6 shows results for the strong accountability algorithm when considering the case described above. The time required by the strong accountability algorithm grows roughly linearly in the number of obligations. In the worst case, the algorithm runs in 65 milliseconds to determine if the set is strongly accountable. In general, if the number of obligations generated by the finite repetitive obligations is not too large (when compared with the original set), the time to compute the algorithm is almost not affect by the repetitive obligations. As algorithm 13 can unroll the repetitive obligations in a trivial way, the overhead of this procedure will be small considering the number of repetitive obligations is small. Figure 6.7 presents the execution times for computing strongly accountability under finite repetitive obligations when the whole set is not strongly accountable. As previous case, we can see that computing strongly accountability for a non strongly accountable set is a little bit faster than computing for a strongly accountable set for obvious reasons.

Finite and Infinite repetitive obligations, and cascading of obligations For these empirical evaluations, we add one obligation to a strongly accountable obligation set. This obligation generates 1000 new obligations. We also add 50 finite repetitive obligations to a strongly accountable obligation set. These 50 obligation in turn is going to generate 1000 new obligations. Finally, we add 5 infinite cascading of obligations, these generate more 1000 obligations The algorithm checks whether all this set together are strongly accountable.

Figure 6.8 shows results for the strong accountability algorithm when considering the case described above. The time required by the strong accountability algorithm grows roughly linearly in the number of obligations. In the worst case, the algorithm runs in 120 milliseconds to determine

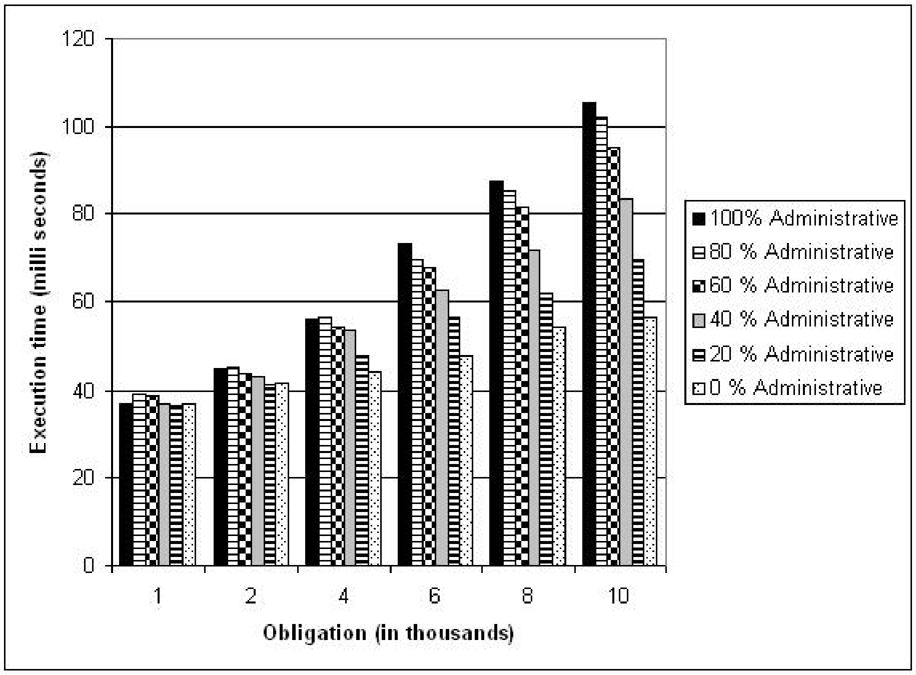

Figure 6.4: Execution time vs. number of obligations (directly cascading of obligations)

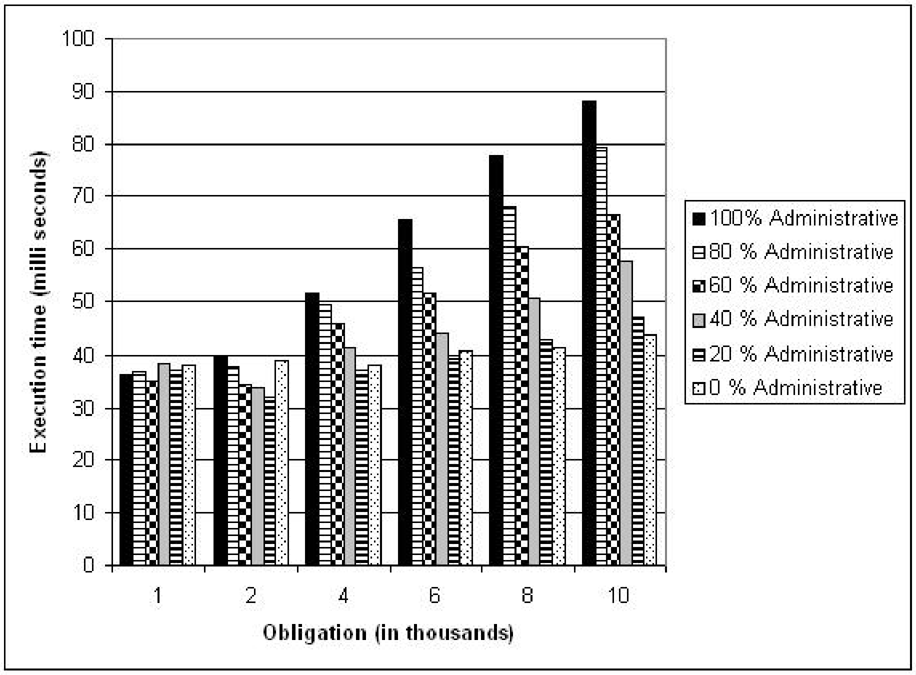

Figure 6.5: Execution time vs. number of obligations (directly cascading of obligations, non-strongly accountable set)

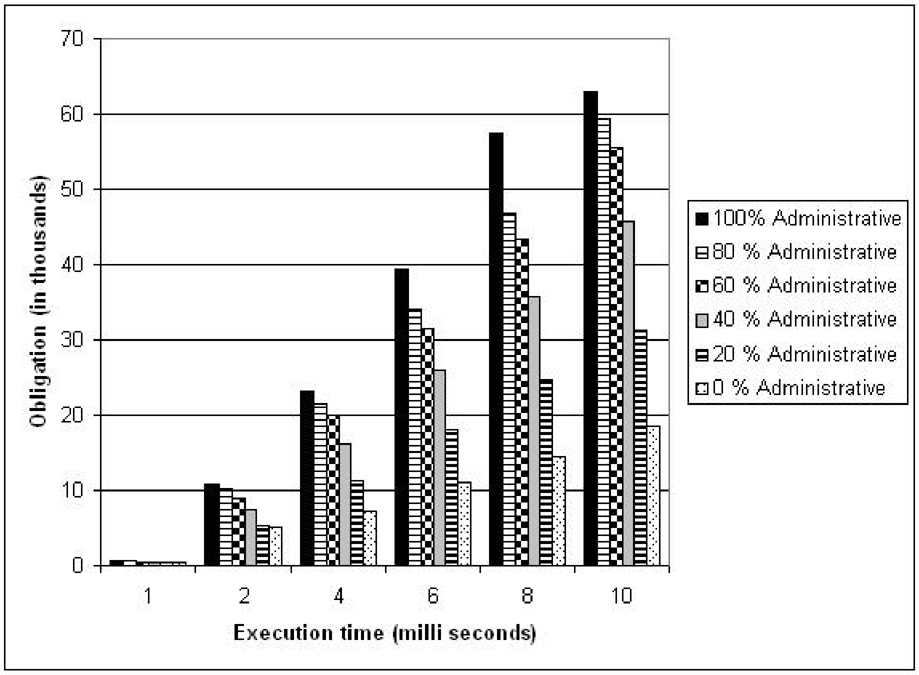

Figure 6.6: Execution time vs. number of obligations (finite repetitive obligations)

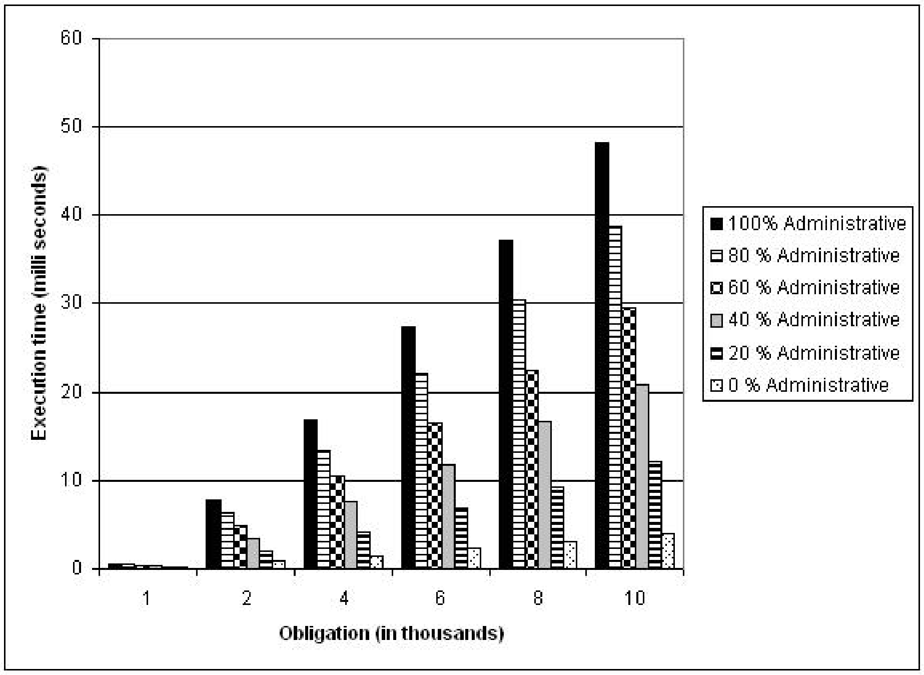

Figure 6.7: Execution time vs. number of obligations (finite repetitive obligations, non-strongly accountable set)

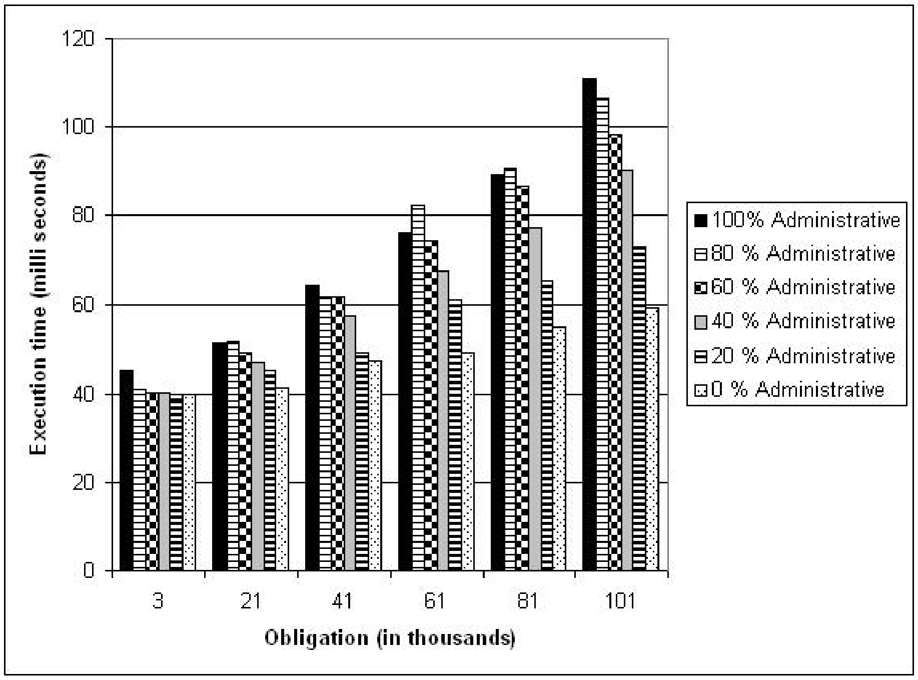

Figure 6.8: Execution time vs. number of obligations (finite/infinite repetitive obligations and directly cascading of obligations)

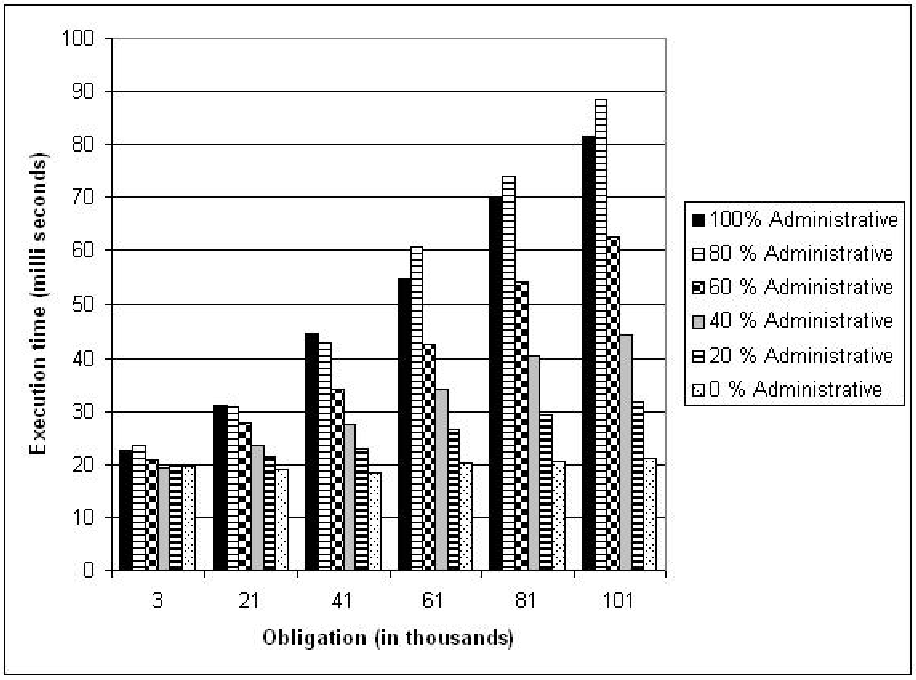

Figure 6.9: Execution time vs. number of obligations (finite/infinite repetitive obligations and directly cascading of obligations, non-strongly accountable set)

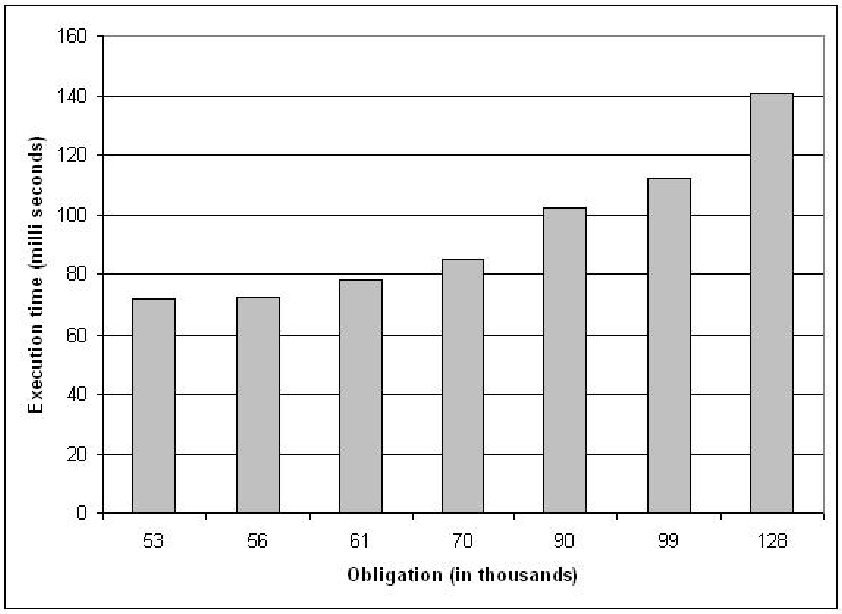

Figure 6.10: Execution time vs. number of obligations

if the set is strongly accountable. The performance of this algorithm is similar to the algorithm that tries to decide strongly accountability under pure cascading. This is due to the fact that unfolding the pure cascading obligation dominates the cost of unfolding infinite and infinite repetitive obligations (considering the three sets are generating the same number of obligations). When comparing this algorithm with the incremental algorithm for computing strongly accountability in chapter 3, we can see this algorithm again needs double the time to decide strongly accountability when not considering cascading. Figure 6.9 presents the execution times for computing strongly accountability for the above case, when the obligation set is not strongly accountable. As we can see from the figure, in the worst case, the algorithm needs less than 90 milliseconds.

Increasing the number of cascading obligations In this empirical evaluation, we keep the number of non-cascading obligation fixed (5000 obligations). And all the obligations are administrative. Then, we vary the number of cascading obligations (finite, infinite repetitive, and pure cascading) from $3,000$ to $76,000$. Figure 6.10 presents the results. We can see the time grows linearly when increasing the number of the cascading of obligations. Even, for more than $126,000$

obligations the algorithm can decide strongly accountability in less than 140 milliseconds.

6.7 Summary

This chapter has presented a taxonomy of cascading of obligations. It also has presented a proof that shows that deciding strongly accountability in the presence of cascading of obligations for user obligation using mini-RBAC/mini-ARBAC as its authorization states is NP-hard with respect to the policy and obligation pool size. We have also presented decision procedures for checking strongly accountability in the presence of some restricted versions of cascading of obligations. Finally, it has presented some empirical evaluations that shows that one can enforce strongly accountability in the presence of cascading of obligations in less than 120 milliseconds.

CHAPTER 7: RELATED WORK

This chapter discusses the works related to this dissertation. We start our discussion by presenting the differences and similarities between our work and other works in the obligation field. Then, we present some related research about how security can interfere with usability and how we address these issues in our framework. Many of the contributions presented in this dissertation are in the form of policy analysis, thus we present some related work that served as the inspiration to our work. Finally, we discuss the advantages of using obligation systems over workflow managements systems (WFMS). To this end, we present some related work in the are of workflow.

7.1 Obligation Models

Many obligation models have been proposed, ranging from largely theoretical [24, 31, 40] to more practical [26, 32, 38, 41, 54]. Minsky and Lockman [40] were the first to suggest incorporating obligations into authorization models. They proposed a very general model that includes concepts of positive obligations, negative obligations, deadline of obligations, obligations triggered by events, compensatory actions for failed obligations, *etc.* The model is probably too abstract to be implemented in a real world system.

Today, many policy languages can specify the assignment of obligations as part of the policy. Some of these include XACML [2], EPAL [5], KAoS [55] and Ponder [23]. XACML [2] supports only system obligations that are triggered by events and have an immediate deadline. EPAL [5] is somehow similar to XACML and assumes obligations are defined as actions that must be taken by the environment, and that they do not interfere with each other. Ponder, on the other hand, assumes user obligations that depend on authorization, but obligations do not have a time window associated with them, and must be fulfilled immediately.

Bandara *et al.* [11] extended the model proposed by Irwin *et al.* [31] for obligations in order to create more complex policy rules. The language proposed is more expressive, yet the strong accountability problem remains tractable. However, no accountability decision procedure was presented.

Gama and Ferreira [26] presented an implementation of an obligation model called Heimdall, which is based on policies written in xSPL. Their model can only enforce system obligations. On the other hand, our model deals with user obligations that depend on authorizations. In that sense, their work is complimentary to ours.

The model of Bettini *et al.* [15] uses logic-programming to select policy rules in order to minimize the number of obligations and provisions one must incur. Unlike our model, they consider that an obligatory user always has the necessary permissions to fulfill his obligations.

Dougherty *et al.* [24] presented an abstract obligation model that relates a program execution path with obligations. The model is very expressive and can handle both positive and negative obligations. They also consider repetitive obligations, penalties for violating an obligation and also states for obligations. They do static analysis by using Büchi automata to answer questions such as determining whether two obligations are contradictory or whether a run of the system fulfills a given obligation, *etc.* Thus, their analysis work focuses on system obligations, rather than on user obligations.

Ni *et al.* [41] presented a concrete model for obligations that interact with permissions based on PRBAC [42] that handles repetitive obligations, pre and post-obligations and also conditional obligations. They also presented two algorithms to analyze the dominance and infinite obligation cascading properties. In addition, they presented other important issues that one needs to consider when implementing a real obligation system, namely, techniques for analyzing unfulfillable obligations, sanctions and reward mechanisms. Their obligation model is more expressive than ours. However, they did not address how to decide accountability in such a model. In the same vein, they also did not investigate the impact of unfulfilled obligations. In this regard, we address other problems in this dissertation.

May *et al.* [38] presented a formal model for legal privacy policies of HIPAA. They considered a basic model of obligations without considering many theoretical issues. They used model checking to analyze properties with respect to some fixed policy rules. They did not present any experimental results of the efficiency of their approach.

Swarup *et al.* [54] presented a model for data sharing agreements in which a set of obligations is considered as constraints. They also suggested using model checking to verify some properties in their model, though they did not mention any practical implementation. The nature of the work that we solve in this dissertation is intrinsically different than their work.

Casassa and Beato [17] provided a formal framework to enforce and specify obligation policies. In their model, they allowed user and system obligations. Obligations can be triggered by time or events, and they used a special kind of action called *on violation actions* for restoring the security state of the system when some user obligations are violated. Such actions allow the system to take some counter measures in order to fulfill the missed obligation. By contrast, we give the administrator some tool support in order to handle violated obligations (*e.g.*, finding the responsible user for the missed obligations, finding the obligations that can be affected by the missed obligations, and techniques to restore the accountability of the system).

Katt *et al.* [37] augmented the UCON model [43] to support post-obligations. Their system considers two types of obligatory actions , *non-trusted obligations* and *trusted obligations*. Trusted obligations are performed by the system, so they consider that they are never violated. Non-trusted obligations; however, are user obligations and can be violated. They proposed a mechanism that makes decisions based on the status of fulfillment of the non-trusted obligations (*e.g.*, if a client did not pay a bill, the system needs to send an email to the client). However, they did not consider interaction of authorization systems and obligations.

Hilty *et al.* [29] provide an obligation specification language (OSL) for distributed usage control. They also show how an OSL can be further translated into a language for expressing rights, which can be enforced by some existing DRM mechanisms. In contrast to our work, they consider obligations in a data containment mechanism, whereas we consider obligations and their interactions with authorization systems.

7.2 Security and Usability

When security policies are more complex, access denial may indicate only that the desired action cannot be performed under the current circumstances. Recently, researchers have studied various situations in which understanding and remedying a denial is more difficult. For instance, Kapadia *et al.* [36] have studied the problem of explaining access denial that is based on environmental factors, such as time and location of access attempt. Bauer *et al.* [13] have studied problems that arise in managing authorization state in large scale systems. Cranor and Garfinkel [21] have studied the relationship of security and usability in many practical problems (*e.g.*, phishing, password generation), and how the complexity of correctly configuring security features such as authorization state can lead to user errors or to users turning off security mechanisms altogether. In [22], Cranor *et al.* have addressed the problem of privacy issues when considering usability. Other authors [8] [20] have studied how user actions can affect security mechanisms. In our work, instead of explaining an access denial, we present the user with a plan of action that enables her to perform her desired action. In addition, when the system falls in an unsafe state (*i.e.*, unaccountable), we provide a set of strategies to the system administrator that enables her to restore the system state to a safe state (*i.e.*, accountable).

7.3 Policy Analysis

Stoller *et al.* [53] studied the mini-ARBAC/mini-RBAC policy through experimental analysis on the *role reachability* problem [33, 51]. In this problem, one seeks a sequence of grants and revokes that modifies the current role memberships of a given user to include a given role. Determining whether there is such a sequence that leads to the user having a given role is, in the unrestricted case, PSPACE-complete [51]. This is particularly daunting because RBAC systems used in practice often have hundreds of roles, thousands of users, and millions of objects [25].

The role reachability problem can be reduced trivially to one of the techniques that we propose to use for accountability restoration, namely the action failure feedback problem (see chapter 4). For dealing with a generalized variant of the role-reachability problem that arises in this context, we have developed an AI planning tool that is often able to find suitable action sequences (provided the sequences are not too long) [47].

7.4 Workflows

Nowadays, many workflow management systems (WFMS) are deployed in practice [1, 3, 4]. However, the majority of the systems assume that the users have the permissions they need when they attempt to execute a task. Sometimes this assumption is not realistic and makes it difficult to create more complex security policies, such as separation of duty, Chinese wall, mandatory vacation, job rotation, *etc.* To address this problem, Atluri and Huang [10] have introduced a workflow authorization model that is able to give or to remove permissions to users only when they are executing their tasks. Bertino *et al.* [14] have presented a workflow management system based on RBAC that is able to check dynamic and static constraints. They have also presented an AI planning approach that attempts to generate possible role-user assignments for a given task; nevertheless, their model does not support temporal or event-based constraints. Kandala and Sandhu [35] have proposed workflow models that can support separation of duty constraints. However, their models also do not accept temporal constraints. Instead, they check for separation of duty just at the time the user attempts to take the action. Combi and Pozzi [19] have created a model that helps administrators to allocate users based on some predefined metrics (*e.g.*, time and user skills). In general, WFMS have some notions of obligations embedded in it. Typically, WFMS [14, 19] are composed of a set of individual tasks used to achieve a common goal. Such tasks can be obligatory actions, and in that sense the work presented in this dissertation can be seen as complimentary to the workflow systems. However, we leave the integration of user obligations and WFMS as a future work.

CHAPTER 8: CONCLUSION AND FUTURE WORK

In recent years, with the growth of the Internet and computer networks, more and more organizations started using computers to assist their business processes. It is clear that in their business processes, organizations will impose some obligatory actions for their employees. The interaction of obligations, security policy and business process raise the complexity of managing such environments. Thus, computer aided tools are necessary to assist these organizations in managing their business processes. This dissertation is part of a project investigating authorization systems that assign obligations to users. We are particularly interested in obligations that require authorization to be performed and that, when performed, may modify the authorization state. In this context, a user may incur an obligation she is unauthorized to perform. Prior work [31] has introduced properties of the authorization system state that ensures users will be authorized to fulfill their obligations on time. The properties are called weak and strong accountability. The stronger version of accountability requires that each obligation be authorized throughout its entire time interval. Whereas the weaker version allows an obligation to be unauthorized during part of its time interval, provided that if the obligated user waits for other obligations to be fulfilled, it is guaranteed that the action will become authorized before its deadline. While a reference monitor can mitigate violations of accountability by denying actions that violate accountability, it cannot prevent them entirely. Although, prior work [31] have addressed many issues in user obligation systems (*e.g.*, accountability property, blaming assignment, *etc.*). Still many questions were without answers. To this end, we present a user obligation architecture and identify several issues that appear when user obligations can interact with authorization system. Namely, we are interested in studying the scalability and performance of deciding accountability, how different authorization models can affect the decision of accountability, the balance between security and usability, and finally, how different types of obligations can affect the performance of such a system.

The first challenge that we have studied was to check the impact of adding decision procedures for both weak and strong accountability in a reference monitor. For this, we have presented an algorithm to decide strong accountability and two methods to decide weak accountability. Both procedures are based on an obligation system that uses mini-ARBAC/mini-RBAC as its authorization system. Our experimental results demonstrate that the performance of the algorithm for strong accountability is excellent. On the other hand, the performance of both methods for weak accountability are only adequate for medium to small size problem instances. In fact, we have proved that even using the simple authorization model, mini-ARBAC/mini-RBAC, the weak accountability problem is co-NP complete. Although, the empirical evaluations of our techniques support the thesis that the strong accountability procedure can be added to a reference monitor, it is important to note that some limitations might exist. (i) We did not have access to any real obligations sets and mini-ARBAC/mini-RBAC policies. Thus, our empirical evaluations are based on problem instances that are generated synthetically. We tried our best to construct problem instances that we think would represent real world obligation sets and policies. To this end, we have hand-crafted problem instances, we have also used mini-ARBAC/mini-RBAC policies presented in other works, and we have also non-deterministically generated problem instances. In fact, policy generation and benchmarks are a rich topic and could be consider a hard problem by itself. (ii) In our techniques, we have used the simplified version of ARBAC and RBAC. This simplifies accountability decision, since the simplified authorization model did not support role hierarchies, sessions, changes to permission-role assignments, or role administration operations, such as creation or deletion of roles. We believe greater generalization is also possible, for instance, to support ARBAC models that support role hierarchies and in which the role hierarchy can be modified. However, doing so seems likely to increase the algorithm's complexity by a factor of the number of roles in the system.

Secondly, we have studied the flexibility of our obligation model. To this end, we instantiate our concrete model to use the extended mini-HRU Access Control Matrix Model as its authorization system. We have presented a decision procedure for strong accountability using this obligation model. Our empirical evaluations shows that deciding strong accountability for the extended mini-HRU authorization models is 13 times slower than a model using mini-ARBAC/mini-RBAC as its authorization model. Despite the higher complexity, the algorithm still runs in less than 90 milliseconds. With this comparison, we have tried to show that our obligation model can support other authorization models. In fact, it is our believe that to be used in practice our obligation model needs to support other types of authorization models.

Thirdly, we have presented an approach to solve the *action failure feedback problem* based on AI planning. The action failure feedback attempts to address the balance between the security goal and usability of a system that incorporates explicit management and enforcement of user obligation. Our approach is based on user obligation system that uses mini-RBAC and mini-ARBAC as its authorization system. We have also shown that the *action failure feedback problem* is PSPACE-hard that partially justifies our approach of using an AI planner to solve the problem. We have also presented empirical results that demonstrate that our approach can be useful for small size problem instances. Exploring other AI planners (*e.g.*, GraphPlan, SatPlan) for solving this problem could lead to more efficient solutions. Another interesting future direction is to explore whether imposing some restrictions on the problem would yield any efficient solution. One important issue that arises when someone uses our failure feedback module is that it may be used to compromise the privacy of a system. This is due to the fact that, when a user receives a plan of actions, it might contain actions not for himself, but also for others. Based on these actions, he can infer the system's security policy. Addressing this problem is a matter of future work.

Fourthly, as mentioned before, the idea of preserving accountability as a system invariant sometimes might not be achieved, as users may fail to fulfill their obligations. For this, we have provided some restoration techniques that can be used by the system administrator to restore accountability. We also have introduced different notions of authorization dependency among obligations. We also have provided formal specification and techniques for calculating two different notions of slice that calculates the set of obligations that the administrator needs to consider when applying the different restoration techniques given a set of violated obligations. In this dissertation, we have presented slice definitions, but no algorithm to find them. In fact, the design of an efficient algorithm remains an open problem. At minimum, it should compute the dependence relation in a lazy way. Moreover, minimality in the dependence relation will be less of an issue in a real algorithm, which will focus on constructing the slice, not on the dependence relation that defines it. Closely related to the accountability-restoration problem is the problem of ensuring that the effect of violation and restoration leaves the authorization system in the same state it would have been if the violation had never occurred. The user obligation systems we consider supports two kinds of actions, namely obligatory and discretionary actions. Obligations to perform administrative actions can be used to give permissions to users so that they can perform discretionary actions. For instance, if a new employee needs to be granted appropriate permissions to enable her to perform her job function, restoring accountability by removing obligations to grant those permissions would be unacceptable. Thus, some administrative obligations must not be removed for reasons other than their effect on other obligations. The administrator or manager who has the responsibility to restore accountability must take this into consideration when selecting an accountability restoration strategy. We did not address this in this dissertation, and leave this as future work.

Finally, the last problem that we have addressed in this dissertation was whether it is possible to decide strongly accountability in the presence of cascading of obligations. Thus, we have presented some restricted notions of cascading obligations, and we have shown a polynomial time algorithm to decide strong accountability in presence of them. Our empirical evaluations indicated that depending on the number of new obligations incurred one can decide strongly accountability in less than 100 milli-seconds. For achieving this performance we have assumed obligations are authorized by only one policy rules. It seems highly unlikely that permitting multiple policy rules per action would lead to a tractable accountability decision problem. Other limitation of model is that we can have an explosion in the number of the cascading of obligations. Without any bound on the chain length of the cascading, the accountability problem can become intractable. We have pointed some possible solutions to these problems, we can bound the length of cascading chains, by establishing a partial order on actions, and requiring that obligatory actions incurred must be strictly dominated by the actions that cause them to be incurred. Another possibility is to statically verify the absence of cycles in the policy rules. Or yet, have cascading of obligations only for non administrative actions. In such cases, one does not need to unroll all the non-obligatory obligations. Instead, it is only necessary unfold the non-administrative obligations one period after the last non-cascading obligation.

8.1 Future Work

Decentralized Environment The techniques developed and discussed in this dissertation for deciding and maintaining accountability in a system are based on an assumption that the *Policy Enforcement Point* (PEP) is centralized. This signifies that all access request and incurring of obligations are checked by the centralized reference monitor for permissibility. Now, it is quiet natural to address whether our techniques could be extended to support systems where the PEP is decentralized.

We assume that in a distributed systems multiple autonomous domains collaborate to form a virtual organization. Each of the domains are governed by their own security policy, but can interact with each other (*e.g.*, one domain can assign obligations to other domains). Due to the privacy concerns it is expected that domains do not reveal their security policy totally to other domains. Thus, extending our abstract and concrete model to support obligations in such a distributed environment is not trivial. Facilitating the support of managing and enforcing user obligations in such an environment creates some technical issues that must be addressed. *First*, what changes should be made to the policy of each of the domains so that one domain can assign obligations to other domains. *Secondly*, when one domain ($domain_A$) wants to assign an obligation to another domain ($domain_B$) that violates the accountability property of $domain_B$. The question is how does $domain_B$ explain to $domain_A$ the reason behind the violation of the accountability without revealing its policy. *Thirdly*, how does one determine whether the accountability property is maintained in the distributed system as a whole.

One possible design could be an existence of a trusted third party (TTP) which acts as the centralized PEP for the distributed system. All the *internal* and *external* obligations are maintained by the TTP. For this, TTP needs to have full knowledge about the policies of individual domains of the distributed system. Internal obligations here refer to obligations that are incurred by the policy of the individual domains whereas external obligations are obligations that are assigned by some external domains. In such a design, TTP is the bottleneck and single point of failure. Furthermore, by compromising TTP an attacker can gain access to the policies of each of the domains. Thus, a possible solution would be a design where each domain maintains its obligations (both internal and external) and knowledge of its own policy. The determination of the accountability property of the whole system is determined using an intuition that if each domain is in an accountable state, so then is the distributed system as a whole.

Parametrized ARBAC/RBAC authorizations system The techniques presented in this dissertation were using mini-RBAC and mini-ARBAC as the the authorization state. However, to be practical we may need to extend our techniques to use parametrized version of the ARBAC/RBAC as the authorization state. Doing so, will permit our obligation architecture to express many different types of obligations that exist in real world. For instance, Alice's doctor is obliged to correct her protected health information. Role hierarchy, and parametrized roles can be modeled in our current model without significant modifications.

Support for other types of obligations In this dissertation, we have considered only positive user obligations triggered by events. One possibly future direction could be relaxing this restriction and fully support other types of obligations, namely, negative obligations, conditional obligations, pre-obligations, incremental obligations, *etc.* Several changes needed to be made in our concrete model and algorithms to deciding accountability in order to support these new types of obligations. It going to be necessary to modify how obligations are incurred in our system. Obligations will need to have states, and in each clock tick the system would need to check whether they changed or not. Another important issue is how to define accountability in the presence of negative obligations. An obligation system can enforce a negative obligation by denying a user to execute the action defined in the negative obligations. Or, the obligation system can just monitor any violations of negative obligations. In our notion of obligation, there is no explicit representation of an obligation's purpose or significance, making it difficult to decide how important an obligation is. So, we leave it in the hand of the administrator to decide which obligations to remove when restoring accountability. We can extend our model in order to support purpose of obligations. This will allow for more efficient automatic tools for restoring accountability, as well as for user allocation.

Functional dependencies In our model, we only consider authorization dependencies. In order to model a richer obligation system, one might need also to consider functional dependencies among obligations. Recall that the creation of obligations in our model are governed by policy rules that have the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. A possible way to introduce functional dependencies in our model is to hard-code them into the F_{obl} functions.

BIBLIOGRAPHY

[1] IntalioCloud. http://www.intalio.com/. [Online; accessed July-2010].

[2] OASIS eXtensible Access Control Markup Language (xacml). http://www.oasis-open.org/committees/xacml/. [Online; accessed May-2009].

[3] ProcessMaker. http://www.processmaker.com/. [Online; accessed July-2010].

[4] Websphere MQ Workflow. http://www-01.ibm.com/software/integration/wmqwf/. [Online; accessed July-2010].

[5] Enterprise privacy authorization language (EPAL) version 1.2, November 2003. http://www.zurich.ibm.com/pri/projects/epal.html.

[6] Cadence SMV, 2009. http://www.kenmcmil.com/.

[7] Description of the policies used in the experiments, 2009. https://galadriel.cs.utsa.edu/policies/.

[8] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.

[9] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2nd edition, 2008.

[10] Vijayalakshmi Atluri and Wei kuang Huang. An authorization model for workflows. In *ESORICS '96: Proceedings of the 4th European Symposium on Research in Computer Security*, pages 44–64, London, UK, 1996. Springer-Verlag.

[11] Arosha Bandara, Jorge Lobo, Seraphin Calo, Emil Lupu, Alessandra Russo, and Morris Sloman. Toward a Formal Characterization of Policy Specification Analysis. In *Annual Conference of ITA (ACITA), University of Maryland, USA*, September 2007.

[12] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. *Security and Privacy, IEEE Symposium on*, 0:184–198, 2006.

[13] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. Real life challenges in access-control management. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 899–908, New York, NY, USA, 2009. ACM.

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

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

[16] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[17] M. Casassa and F. Beato. On Parametric Obligation Policies: Enabling Privacy-Aware Information Lifecycle Management in Enterprises. In *Policies for Distributed Systems and Networks.*, pages 51 –55, jun. 2007.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[19] Carlo Combi and Giuseppe Pozzi. Task scheduling for a temporalworkflow management system. In *TIME '06: Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning*, pages 61–68, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Lorrie Faith Cranor. A framework for reasoning about the human in the loop. In *UPSEC'08: Proceedings of the 1st Conference on Usability, Psychology, and Security*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[21] Lorrie Faith Cranor and Simson Garfinkel, editors. *Security and Usability*. O'Reilly Media, 2005.

[22] Lorrie Faith Cranor, Praveen Guduru, and Manjula Arjula. User interfaces for privacy agents. *ACM Trans. Comput.-Hum. Interact.*, 13(2):135–178, 2006.

[23] 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.

[24] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In *CESORICS '07: Proceedings of the 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, pages 375–389, 2007.

[25] Michael P. Gallaher, Alan C. Oconnor, and Brian Kropp. The Economic Impact of Role-Based Access Control, March 2002. Available at http://www.nist.gov/director/prog-ofc/report02-1.pdf.

[26] 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.

[27] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19:461–471, August 1976.

[28] Health Resources and Services Administration. Health insurance portability and accountability act, 1996. Public Law 104-191.

[29] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In Joachim Biskup and Javier Lopez, editors, *Computer Security - ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer Berlin, Heidelberg, 2008.

[30] Keith Irwin. *A System for Managing User Obligations*. PhD thesis, North Carolina State University, 2008.

[31] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

[32] Keith Irwin, Ting Yu, and William H. Winsborough. Assigning responsibilities for failed obligations. In *iTrust '08: IFIPTM Joined iTrust and PST Conference on Privacy, Trust Management and Security*, pages 327–342. Springer Boston, 2008.

[33] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic error finding in access-control policies. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 163–174, New York, NY, USA, 2011. ACM.

[34] A. J. I. Jones. On the relationship between permission and obligation. In *ICAIL '87: Proceedings of the 1st international conference on Artificial intelligence and law*, pages 164–169, New York, NY, USA, 1987. ACM.

[35] Savith Kandala and Ravi Sandhu. Secure role-based workflow models. In *Das'01: Proceedings of the fifteenth annual working conference on Database and application security*, pages 45–58, Norwell, MA, USA, 2002. Kluwer Academic Publishers.

[36] 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.

[37] Basel Katt, Xinwen Zhang, Ruth Breu, Michael Hafner, and Jean-Pierre Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132, New York, NY, USA, 2008. ACM.

[38] Michael J. May, Carl A. Gunter, and Insup Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 85–97, Washington, DC, USA, 2006. IEEE Computer Society.

[39] L.T. McCarty. Pemissions and obligations. In *Proceedings IJCAI-83*, 1983.

[40] Naftaly H. Minsky and Abe D. Lockman. Ensuring integrity by adding obligations to privileges. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 92–102, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[41] Qun Ni, Elisa Bertino, and Jorge Lobo. An obligation model bridging access control policies and privacy policies. In *SACMAT 2008: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 133–142, New York, NY, USA, 2008. ACM.

[42] Qun Ni, Alberto Trombetta, Elisa Bertino, and Jorge Lobo. Privacy-aware role based access control. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 41–50, New York, NY, USA, 2007. ACM.

[43] Jaehong Park and Ravi Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

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

[45] Murillo Pontual, Omar Chowdhury, William Winsborough, Ting Yu, and Keith Irwin. Toward practical authorization-dependent user obligation systems. In *Proceedings of the 5th International Symposium on ACM Symposium on Information, Computer and Communications Security (ASIACCS'2010)*, pages 180–191. ACM Press, 2010.

[46] Murillo Pontual, Omar Chowdhury, William H. Winsborough, Ting Yu, and Keith Irwin. On the management of user obligations. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 175–184, New York, NY, USA, 2011. ACM.

[47] Murillo Pontual, Keith Irwin, Omar Chowdhury, William H. Winsborough, and Ting Yu. Failure feedback for user obligation systems. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 713–720, Washington, DC, USA, 2010. IEEE Computer Society.

[48] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[49] Ravi S. Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the First ACM Workshop on Role-Based Access Control*, 1996.

[50] 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.

[51] Amit Sasturkar, Amit Yang, Scott D. Stoller, and C.R. Ramakrishnan. Policy analysis for administrative role based access control. In *Computer Security Foundations Workshop, IEEE*, volume 0, pages 124–138, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[52] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2004.

[53] Scott D. Stoller, Ping Yang, C R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 445–455, New York, NY, USA, 2007. ACM.

[54] Vipin Swarup, Len Seligman, and Arnon Rosenthal. A data sharing agreement framework. In *Information Systems Security, Second International Conference, ICISS 2006, Kolkata, India, December 19-21, 2006, Proceedings*, pages 22–36, 2006.

[55] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[57] A. Witten 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.

[58] 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.

VITA

Murillo Pontual received B.S. and M.Sc. in Computer Science at the Federal University of Pernambuco in Recife, Brazil in 2003 and 2005, respectively. He got his M.Sc degree with a thesis about Secure multi-party computation for privacy-preserving Linear Algebra and Privacy-Preserving Statistical Computations. This is his third year as a Ph.D student working under the supervision of Dr. William H. Winsborough at the University of Texas at San Antonio, and he is currently working in the design and implementation of practical user obligation systems. He has worked as web programmer in W3 Tecnologia company, network administrator in the Federal University of Pernambuco, and instructor in Santa Maria University, and Unibratec Technical School (both in Brazil). His professional objective is to get a faculty position in the field of computer security.