Assigning Responsibility for Failed Obligations

Keith Irwin, Ting Yu and William H. Winsborough

Abstract Traditional security policies largely focus on access control. Though essential, access control is only one aspect of security. In particular, the correct behavior and reliable operation of a system depends not only on what users are permitted to do, but oftentimes on what users are required to do. Such obligatory actions are integral to the security procedures of many enterprises. Unlike access control, obligations assigned to individual users are often unenforceable, that is, the system cannot ensure that each obligation will be fulfilled. Accurately determining who was at fault when obligations are not met is essential for responding appropriately, be it in terms of modified trust relationships or other recourse. In this paper, based on a formal metamodel of obligations, we propose an approach for fault assessment through active online tracking of responsibilities and dependencies between obligations. We identify and formalize two key properties for the correct assessment of fault, and design responsibility assignment and fault assessment algorithms for a concrete yet general access control and obligation system.

1 Introduction

A security policy defines the correct behavior of an information system. Today, a majority of techniques, literature, and infrastructure in security policy management focus on access control. Though essential, access control is only one aspect of security. In particular, the correct behavior and reliable operation of a system relies not only on what users are permitted to do, but oftentimes on what users are required to do. Such obligatory actions are integral to the security procedures of many enterprises. For instance, when an employee leaves a firm, it is usually very important that the employee's computer accounts and physical access be deactivated. Obligations are also essential to privacy policies. Enterprises that collect or use the private information of individuals must abide by laws that require certain actions, such as deleting data after a certain period of time. Finally, oblgations are often key elements of contracts and other agreements.

Keith Irwin and Ting Yu
North Carolina State University, e-mail: {kirwin,tyu}@ncsu.edu

William H. Winsborough
University of Texas at San Antonio, e-mail: wwinsborough@acm.org

Recently, we have witnessed an increasing trend to express obligations explicitly as part of security policies [5, 10, 14, 18, 19]. Obligation policies have different properties from access control policies. In particular, obligations assigned to individual users, though monitorable (i.e., a system can determine whether and when an obligation is fulfilled), are, in general, not enforceable (i.e., the system cannot ensure that an obligation is always fulfilled). Thus, the handling of failed obligations is an indispensable part of obligation management. Besides conducting necessary actions to compensate for failed obligations, a system should appropriately assign culpability to responsible parties. In this paper, we call this process *fault assessment.*

Fault assessment can be significant for several reasons. It may be a simple question of evaluating the performance of employees at their assigned tasks. However, an obligation failure may result in the violation of a contract, which may lead to sanctions against the responsible party or the organization he represents. Even when an agreement does not rise to the level of a legal contract, failing to meet one's obligations has and should have consequences for one's reputation and the level of trust that others place in one.

At a first sight, the solution to the fault assessment problem seems straightforward. Indeed, in many cases, if a user does not complete her obligation, she is to blame for the failure. On the other hand, it is common that one or more obligations provide necessary privileges or resources to enable the fulfillment of other obligations. For example, in work-flow systems, frequently one user's obligation can be fulfilled only if some other users first complete their obligations, thus making the necessary resources available. A dependency between two obligations occurs if one user is obligated to grant an authorization necessary to enable another user to fulfill his obligation. Fairness requires that if a user cannot fulfill an obligation due to a lack of sufficient privileges or necessary resources, the user should not be considered to be at fault. A lack of privileges may be caused by other users failing to fulfill their obligations. Clearly, in such cases, fault assessment requires analyzing and maintaining the dependencies between obligations, which is a non-trivial task.

In this paper, we propose an active approach to fault assessment through online tracking of responsibilities and dependencies between obligations. The contributions of this paper include:

- Instead of relying on postmortem analysis of the dependency between failed obligations, we propose a framework that allows a system to dynamically observe dependencies at the time when obligations are assigned to users and assign responsibility at that point. We show that such online responsibility tracking significantly reduces the complexity of fault assessment of failed obligations.
- Fault assessment is often application or policy dependent. Instead of advocating a single solution, we identify and formalize two key properties for responsibility tracking based on a metamodel of obligation and access control systems. We argue that any responsibility tracking algorithm should satisfy these two properties in order to achieve correct blame assignment.
- We present an algorithm that computes fault assessment when given a set of failed obligations and their resposibility graph maintained by a system.

- We further instantiate the metamodel and study fault assessment in a concrete access control and obligation system based on an access-matrix model. We show that obligation dependency can be efficiently identified in this system. We propose a generalized obligation responsibility-tracking algorithm, and prove that it satisfies the key properties for responsibility tracking that we identify in the contribution mentioned above.

The rest of the paper is organized as follows. In section 2, we discuss work closely related to this paper. In section 3 and 4, we briefly describe the construction of a metamodel of access control and obligation systems, and introduce the concept of accountability, a security objective for obligation systems. The fault assessment problem and the two properties for responsibility tracking are articulated in section 5. Our discussion of fault assessment in a concrete obligation and access control system is presented in section 6. We conclude in section 7.

2 Related Works

Obligations are common features of security policies in a variety of application domains. For example, due to requirements from laws and other regulations, enterprise privacy policies often include obligations regarding the handling of private information collected from individuals. Typical examples concern the retention and deletion of customers' personal and transactional information, notification when such information is shared with other parties, and auditing after access to private information [12, 13]. Obligations are also integral to the security policies of cross-domain data sharing, since data sharing and handling contracts between autonomous entities, such as responsive forwarding and nondisclosure agreements and usage notification, are often specified in the form of obligations [17].

Several policy languages have been proposed that support the specification of obligations in security policies. XACML [18] and KAoS [19] both have a limited model of obligations. Specifically, they model obligations assigned to a system and cannot describe user obligations, i.e., obligations assigned to ordinary users who are not always trusted to fulfill obligations. Ponder [5], SPL [14], and Rei [10] all support the specification of user obligations.

To the best of our knowledge, Bettini et al. [1] were the first to investigate the analysis of obligations in the context of access control. They studied the problem of choosing appropriate policy rules to minimize the provisions and obligations that a user incurs in order to take a certain action. However, they assume actions in obligations are not subject to access control, and thus they can always be fulfilled. Bettini et al. [2, 3] further extended their work to place under policy control the handling of obligation violations.

Heimdall [6] is a prototype obligation monitoring platform which tracks pending obligations. It detects when obligations are fulfilled or violated. This requires the modeling of time constraints in obligations, which are explicitly supported in its policy language xSPL. Sailer and Morciniec [15] propose a means of using a third party to monitor obligation compliance in contracts in a web services settings.

Hilty, et al. [8] describe how to formally model obligation policies and how to enforce obligation policies which might initially appear to be unable to be monitored. Their work is complimentary to ours.

Responsibility assignment is also related to auditing systems, which collect audit data and analyze them to discover security violations [16]. Most works in auditing target intrusion detection, which is distinct from the problem addressed in this paper.

Obligations and contracts are central concepts in collaborative multi-agent systems. Deontic logic [11] is commonly used to express the agreed obligations among agents. Works in this area typically do not concern the interaction between obligations and access control. Instead, they focus on expressing the dependencies between obligations, assuming such dependencies have already been identified [4, 7]. In this paper, we formalize the key properties for correct identification of dependencies between obligations.

3 Metamodel

We now present a highly abstract model of an obligation system, which we call a metamodel. Any concrete model instantiates one or more of its features. This metamodel was first presented in [9] and more detail can be found there.

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

- $\mathscr{T}$: a countable set of time values.
- $\mathscr{S}$: a set of subjects.
- $\mathscr{O}$: a set of objects with $\mathscr{S} \subseteq \mathscr{O}$.
- $\mathscr{A}$: a finite set of actions that can be initiated by subjects. Each action is a function that takes as input the current system state (defined just below), the subject performing the action, and a finite sequence of zero or more objects. It outputs a new system state.
- $\mathscr{B} = \mathscr{S} \times \mathscr{A} \times \mathscr{O}^* \times \mathscr{T} \times \mathscr{T}$: a set of obligations that can be introduced to the system. Given an obligation $b \in \mathscr{B}$, we use $b.s$ to refer to the subject, $b.a$ for the action, $b.O$ for the objects that are parameters to the action, and $b.t_s$ and $b.t_e$ to refer to the start and end times.
- $\mathscr{S}\mathscr{T} = \mathscr{T} \times \mathscr{F}\mathscr{P}(\mathscr{S}) \times \mathscr{F}\mathscr{P}(\mathscr{O}) \times \Sigma \times \mathscr{F}\mathscr{P}(\mathscr{B})$: the set of system states. Here, we use $\mathscr{F}\mathscr{P}(\mathscr{X}) = \{X \subset \mathscr{X} | X \textit{ is finite}\}$ to denote the set of finite subsets of the given set. We use $st = \langle t, S, O, \sigma, B \rangle$ to denote systems states, where t is the time in the system, S and O are the subjects and objects currently in the system, B is the set of pending obligations, and σ is a fully abstract representation of all other features of the system state. Σ, the domain of abstract states, is possibly

infinite[1]. We use $st_{cur} = \langle t_{cur}, S_{cur}, O_{cur}, \sigma_{cur}, B_{cur} \rangle$ to denote the current state of the system.

- $\mathscr{P}$: a set of policy rules. Each policy rule specifies an action that can be taken, under what circumstances it may be taken, and what obligations (if any) results from that action. Each policy rule p has the form $p = a(st, s, O) \leftarrow cond : F_{obl}$ in which $a \in \mathscr{A}$ and $cond$ (denoted by $p.cond$) is a predicate in $\mathscr{S} \times \mathscr{T} \times \Sigma \times \mathscr{O}^* \rightarrow \{true, false\}$, indicating that subject s is authorized to perform action a on objects O at time t with the system in state σ if $cond(s, t, \sigma, O)$ is true. F_{obl} is an *obligation function*, which takes the current state of the system σ, the current time, the subject s, and the arguments O as its input and returns a finite set $B \subset \mathscr{B}$ of obligations resulting from the action. Obligations in B may not be incurred by the same subject who performed the action.

We assume that actions scheduled for a given time can be finished in a single clock tick, and their effect will be reflected in the state of the next clock tick.

Suppose a (finite) set of actions are attempted at the same time, $t = i$, in state st_i. We denote such a set by $AP \subset \mathscr{S} \times \mathscr{A} \times \mathscr{O}^*$. ($AP$ stands for action plan.) The order in which the elements of AP are executed is given by a fixed, arbitrary total order over $\mathscr{S} \times \mathscr{A} \times \mathscr{O}^*$. This means that two actions, if present in AP, will be executed in the same order regardless of other actions that may or may not be in the set. Let us assume $|AP| = n$ and $ap_0, ap_1, \ldots, ap_{n-1}$ enumerates AP in the order mentioned above.

Given st_i and AP, we let the function $\mathsf{apply}(AP, st_i) = st_{i+1}$ in which st_{i+1} is obtained by ordering and applying the actions. Details of how this would be carried out can be found in the previous paper. This defines the *transition* from st_i to st_{i+1} determined by AP. Given st_i, AP, and $ap \in AP$, we let $\mathsf{permitted}(ap, AP, st_i) = true$ if ap is permitted when it is attempted. Otherwise, $\mathsf{permitted}(ap, AP, st_i) = false$.

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

4 Accountability

Since obligations are unenforceable, a system can never guarantee that an obligation will be fulfilled. What a system *can* seek to ensure is that all obligations *could* be fulfilled if the obligated user is diligent. Roughly speaking, what we want is that the only reason that an obligation will go unfulfilled is due to negligence on the part of a user, not because of insufficient privileges or resources.

If performing a requested action would cause some user to incur an obligation they could not fulfill, the system should deny that action. Conversely, if the user will

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

have sufficient privileges to fulfill the obligation, then the system should allow the requested action. However, it is not obvious what the appropriate behavior should be if the ability of the user to perform the obligation depends on whether or not other actions are taken which change his privileges.

In [9], we propose a concept we call *accountability* as a more satisfactory obligation-security notion. Intuitively, if all users will have sufficient privileges and resource to carry out their obligations provided every other user diligently carries out his or her obligations (and no other actions are performed), then we say the system is in an *accountable state*. Starting from an accountable state, if no actions are initiated other than existing obligations being diligently carried out, then the first obligation that becomes violated must be due to lack of diligence on the part of the obligated subject, and that subject should be assigned the blame.

Notice that if at some point the system is in an accountable state and then transitions to a new state through diligent obligation fulfillment, the new state is also accountable. However, when actions are requested that are not required by an existing obligation, to remain in an accountable state, the system needs to analyze the would-be effect of the action on accountability. Once the system determines whether the resulting state will be accountable or not, it can permit or deny the action accordingly.

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

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

Definition 2. A state st is *strongly accountable* if there exists no sequence of valid obligation-abiding transitions that lead st to a type-1 undesirable state.

5 Fault Assessment

An aspect of obligation systems which we wish to automate is the assessment of fault. When an obligation failure occurs, we wish to know which party is at fault for that failure. The first question to examine is whether or not the user to whom the obligation was assigned possessed the necessary privileges and resources to carry out the obligation. If he did, then he is at fault for the failed obligation. If, however, he did not, it is likely that someone else is at fault.

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

Instead, the fault lies with one or more other users for failing to fulfill their own obligations. We do not, however, assume that the system achieves an accountable state all the time. Because of the nature of obligations, we do assume that the system is making some attempt to achieve an accountable state, but the fault analysis does not depend on such a state ever being achieved.

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

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

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

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

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

What we propose is a module in an obligation management system which keeps track of which obligations bear responsibility for which other obligations. Essentially, the module will keep a directed graph of responsibility, indicating which obligations are responsible for which other obligations. As obligations are fulfilled and discharged, and as other circumstances change, this module should update the graph to reflect changes in responsibility.

The responsibility information in the module will serve as a means of determining fault both after the fact and before the fact. For example, users can consult the module to better understand the implications of their actions, in case there are circumstances where they only have time to fulfill one of the obligations assigned to them.

Because, as we demonstrated above, there are a variety of different possible intents behind the assignment of obligations, we wish to allow for a policy which describes what the assignment of responsibility should be. In other words, this policy specifies, given an obligation b, what other obligations are considered responsible to provide the necessary privileges and resources to b.

Such a responsibility assignment policy is application specific. In practice, we would like to have policies that consist of one general rule and some special-case exceptions. That way, instead of having to imagine every possible situation and write a policy for it, the administrator could choose a default policy and then worry about specifying more specific policy rules only for exceptional cases. For example, if both obligations b_1 and b_2 provides the same needed privileges for obligation b, the administrator may determine that in general the early one of b_1 and b_2 is responsible for b. The administrator may further specify the special situations where this general rule does not apply.

5.1 Desirable Properties

Although we have argued that fault assessment is not simply about knowing which obligations enable which other obligations, the dependencies between obligations play an important role in determining fault. Intuitively, if two obligations are entirely independent of each other, then the failure of one should not be blamed on the failure of the other in any circumstance. Similarly, if one is completely, directly, and solely dependent on another obligation, then clearly the failure of the later one should be blamed on the failure of the earlier one.

These two properties effectively form an upper and lower bound for dependency relationships between pairs of obligations. We do not expect that the majority of pairs of obligations will have one of these two properties. Rather we expect that most will live in the great grey area in the middle, where responsibility is unclear. This is why we later describe the specifics of a policy-driven fault assessment engine. But for pairs of obligations which do have one of these two properties, obviously our system should properly assign responsibility or the lack of it. We say a responsibility assignment policy is *valid* if the above two properties are preserved. As such, we aim to formalize the two properties so that we can prove that a given policy is valid. For this purpose, we introduce the following notations to facilitate our discussion.

Let us assume that we have some set of obligations B, a start time, t_0, and a set of possible future times, T, such that for all $t \in T, t > t_0$. For convenience below we augment each obligation $b = obl(s,a,O,t_s,t_e) \in B$ with a version of the function $\mathsf{permitted}$, denoted $b.\mathsf{permitted}$, that is specialized to b. The type of this function is $b.\mathsf{permitted} : \mathscr{F}\mathscr{P}(\mathscr{S} \times \mathscr{A} \times \mathscr{O}^*) \times \mathscr{S}\mathscr{T} \rightarrow \{true, false\}$. Given a set of actions AP to be executed in state st such that $(b.s, b.a, b.O) \in AP$, $b.\mathsf{permitted}(AP, st) = \mathsf{permitted}((b.s, b.a, b.O), AP, st)$.

Definition 3. A *schedule* of obligations in B is a function $H : B \rightarrow T \bigcup \{null\}$ in which $H(b) = t$ means that b is performed at time t. If $H(b) = null$ for some obligation $b \in B$, this means that in schedule H, b is not fulfilled.

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

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

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

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

5.1.1 Not Responsible For

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

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

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

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

5.1.2 Definitely Responsible For

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

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

1. For all valid schedule H defined over B such that $H(b_1) \neq null$, let $H' = H \sim H^{\{b_1\},null}$. Then for all t' such that $b_2.t_s \leq t' \leq b_2.t_e$, $b_2.\mathsf{permitted}(st_{H'}(t'), AP(H', t'))$ is false.
2. There exists some valid schedule H defined over B where $H(b_1) \neq null$, such that for all t' such that $b_2.t_s \leq t' \leq b_2.t_e$, $b_2.\mathsf{permitted}(st_H(t'), AP(H, t'))$ is true.

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

5.2 Fault Assessment

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

If our assignments of responsibility are reasonable, then it should be the case that any obligation depends only on obligations which are before it. It is worthwhile to note that if this property holds, then it will be the case that our graph is acyclic. This is a desirable property since if our graph contains cycles, we could wind up blaming an obligation's failure for causing itself to fail, which is not quite sensible.

When there is an obligation failure, we can then use this graph to determine who was responsible for the failure. In order to do so, we use an algorithm which traces backwards following the links of responsibility in reverse, in order to figure out which failed obligations are to blame for this failure.

The algorithm uses a working set, K, which contains obligations which are potentially at fault and produces an output set, L, of obligations which are at fault. The initial failed obligation is designated as b. The algorithm assumes the existence of sufficient logs to check things such as what the system state, st, was at different times, which we will designate as $st(t)$, and whether or not particular obligations were carried out. The action plan which was executed at any particular time t is represented as $AP(t)$.

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

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

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

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

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

5.3 Responsibility Assignment

Given a set of obligations, we wish to analyze them and form some assignment of responsibility. There is a great deal of flexibility in assigning the responsibility, but there are three properties which the assignment should meet. The first one is that no obligation should depend on an obligation which comes after it. The second two are based on our properties above. If, at the time that we are assigning responsibility, b_1 is not responsible for b_2 according to the definition presented in the previous section, then b_1 should not be assigned responsibility for b_2. If, at the time that we are assigning responsibility, b_1 is definitely responsible for b_2 according to the definition presented in the previous section, then b_1 should be assigned responsibility for b_2.

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

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

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

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

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

Intuitively, each set in $N(b)$ is a set of obligations which collectively supply some needed permission or resource. Once we have a set of subsets of B which has such a property, then we are going to choose precisely one member of each subset to be considered responsible. Because there may be some subsets which overlap, it should be noted that the number of subsets only forms an upper bound for the number of responsible obligations. There are a number of different ways in which the particular member can be selected, and these reflect different policies, as we discuss later.

However, whatever choice we make, our choice will satisfy the three properties outlined earlier. The proofs that this is the case have been omitted for reasons of space. However, this still leaves the question of how efficiently $N(b)$ can be found. As the determination of $N(b)$ is system specific, we cannot argue that it will be efficient in all systems, however we believe that it will be able to be computed efficiently in many real-world systems. In section 6, we demonstrate a system in which it is possible to follow our algorithm because the $N(b)$ can be computed efficiently.

As we have shown above, in our algorithm there is a lot of room for flexibility concerning how we select the specific obligation which is considered to be responsible. So long as at least one obligation from every set in $N(b)$ is selected, then the system is guaranteed to satisfy the principles outlined. As such, we can use any policy we want to select the obligations, for example, oldest obligation, smallest covering set, or highest ranking user, and still be guaranteed that the properties will hold.

6 A Concrete Example

Lets us now consider how fault assessment would work in a more concrete system. Specifically, we will use a system based on the Access Matrix Model. We first introduced this system in [9], but we will summarize it here.

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

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

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

6.1 Responsibility Assignment

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

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

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

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

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

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

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

7 Conclusion

In this paper, we introduce the problem of blame assignment in obligation management, and discuss why straightforward approaches to blame assignment are not feasible. We present an alternate general approach to blame assignment and formalize two properties which any blame assignment algorithm should meet. We further present a general algorithm for assigning blame, and prove that it has the requisite properties. We presented a number of different specific policies which could be used with the general algorithm, and demonstrate that the algorithm was workable on at least some realistic real-world systems by demonstrating how it would work on one particular system which instantiates the metamodel.

This work is part of a larger project to develop a comprehensive study of obligation-management systems. Assuming users are diligent, we already know how, in certain kinds of systems, to preserve accountability. With this paper we also know how to assign blame when users are not diligent. In the future we plan, among other things, to study methods of restoring the system to an accountable state when users are not diligent.

References

1. C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *VLDB*, Hong Kong, China, Aug. 2002. IEEE Computer Society Press.

2. C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Obligation monitoring in policy management. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society Press.
3. C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Network Syst. Manage.*, 11(3):351–372, 2003.
4. H. Chockler and J. Halpern. Responsibility and Blame: A Structural-Model Approach. *Journal of Artifical Intelligence Research*, 22:93–115, 2004.
5. 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, Jan. 2001. Springer-Verlag.
6. P. Gama and P. 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.
7. D. Grossi, F. Dignum, L. Royakkers, and J. Meyer. Collective Obligations and Agents: Who Gets the Blame? In *International Workshop on Deontic Logic in Computer Science*, Madeira, Portugal, May 2004.
8. M. Hilty, D. A. Basin, and A. Pretschner. On obligations. In *ESORICS*, pages 98–117, 2005.
9. K. Irwin, T. Yu, and W. Winsborough. On the Modeling and Analysis of Obligations. In *ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2006.
10. L. Kagal, T. W. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society.
11. P. McNamara. Deontic Logic, 2006. Stanford Encyclopedia of Phylosophy. Available at http://plato.stanford.edu/entries/logic-deontic/.
12. M. C. Mont. A System to Handle Privacy Obligations in Enterprises. Technical Report HPL-2005-180, HP Labs - Research, 2005. Available at http://www.hpl.hp.com/techreports/2005/HPL-2005-180.html.
13. M. C. Mont and R. Thyne. A Systemic Approach to Automate Privacy Policy Enforcement in Enterprises. Technical Report HPL-2006-51, HP Labs - Research, 2006. Available at http://www.hpl.hp.com/techreports/2006/HPL-2006-51.html.
14. C. Riberiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An Access Control Language for Security Policies and Complex Constraints. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.
15. M. Sailer and M. Morciniec. Monitoring and execution for contract compliance. Technical Report TR 2001-261, HP Labs, 2001.
16. R. Sandhu and P. Samarati. Authentication, Access Control, and Audit. *ACM Computing Survey*, 28(1), Mar. 1996.
17. V. Swarup, L. Seligman, and A. Rosenthal. Specifying Data Sharing Agreements. In *IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, London, Ontario Canada, June 2006.
18. X. TC. Oasis extensible access control markup language (xacml). *http://www.oasis-open.org/committees/xacml/*.
19. A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. 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 *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003. IEEE Computer Society Press.