Obligations and their Interaction with Programs

Daniel J. Dougherty,[1] Kathi Fisler,[1] Shriram Krishnamurthi[2]

[1] Department of Computer Science, WPI
[2] Computer Science Department, Brown University
{dd, kfisler}@cs.wpi.edu, sk@cs.brown.edu

Abstract. Obligations are pervasive in modern systems, often linked to access control decisions. We present a very general model of obligations as objects with state, and discuss its interaction with a program's execution. We describe several analyses that the model enables, both static (for verification) and dynamic (for monitoring). This includes a systematic approach to approximating obligations for enforcement. We also discuss some extensions that would enable practical policy notations. Finally, we evaluate the robustness of our model against standard definitions from jurisprudence.

1 Introduction

Modern society recognizes a strong linkage between rights and responsibilities, or between privileges and corresponding obligations. This connection between rights and obligations carries over to computer systems also: when a system is given the right to perform a certain action, there is often a corresponding obligation on the system's subsequent behavior. For instance, when a system is permitted to access a particular resource, such as a file descriptor, the obligation may vary from having to return that resource, not over-use that resource, or not share that resource. In fact, in computer systems, the mere *attempt* to obtain a right may incur an obligation: for instance, when a password-guarded access is denied, the system may be obliged to log the denial.

This connection between rights and obligations ought to be reflected in our descriptions of access-control policies. A popular modern access control language such as XACML [1] (and similarly, EPAL [2]) repeatedly associates obligations with decisions and provides a syntactic element for specifying them, but then says,

> There are no standard definitions for these actions in version 2.0 of XACML. (`<Obligations>`) elements are returned to the PEP [policy enforcement point] for enforcement.

and provides little further information on the structure or interpretation of obligations.

This paper contributes in three respects: by providing a rich model of obligations; by linking these to program actions; and by defining how to perform verification and monitoring of the resulting systems. Our model is unique in several ways: obligations are stateful entities, reflecting the fact that they can change over time; obligations are linked to program actions while still permitting separate expression of policy from program; and the model is abstract enough to encompass a tremendous variety of obligations, which we demonstrate by employing the taxonomy of a standard American legal reference, Black's Law Dictionary [3].

2 Motivating Examples

Obligations impose many kinds of constraints. Some of these constraints are positive (that a person repay a loan; once a file is read, subsequent messages must be encrypted), while others are negative (that companies not sell email addresses to third parties; once a file is read, no messages can be sent). Some require a single action in bounded time (return a library book within 3 weeks), while others require repeated behaviors (renew a subscription annually). Some carry penalties for violation (interest on late payments), while others bear no repercussion (failing to acknowledge a closed bug report). The following examples are representative of the constraints in realistic systems:

Example 1 (*File check-out and return*) A code versioning tool restricts which developers may check out critical files. Furthermore, at most one developer can have write permission for a critical file at any time. When a developer is given access to a file, she is obligated to eventually check the file back in.

Example 2 (*Logging following access denial*) When a user attempts to access a document for which he lacks the required credentials, every subsequent attempt to access documents by that user must be logged.

Example 3 (*Delegation in a bug tracker*) In a bug-tracking system for a software company, users submit bug reports online. An employee scans each report and delegates responsibility for handling the bug to an appropriate developer. As developers work on bugs, they in turn may delegate the handling to other developers. Each developer is obligated to eventually close every bug report they are assigned to that is not awaiting additional input from some user.

Example 4 (*Incremental payment*) An online store allows customers to accumulate balances in their accounts and pay them off incrementally. Purchasing an item obligates the buyer to pay the seller the purchase price.

A viable model of obligations should capture and support reasoning about all of these examples, yet no prior model does. Example 1 is generally supported, but rarely the others. Example 2 arises on a denied, rather than permitted action. Examples 3 and 4 require obligations to have state. Section 7 provides more detail in evaluating related work using these criteria.

3 Foundational Model

Obligations arise in response to user or program actions. In our model we will assume that actions triggering obligations are governed by access-control policies, though this assumption can be relaxed with only minimal changes to our formalism. We treat obligations as constraints on future behavior, and do not address provisions (constraints on current or past behaviors) except insofar as these are used in access control. We describe our model of systems and obligations abstractly, to accommodate a wide variety of concrete implementations.

We represent program executions by infinite sequences π of state/action pairs, or *paths*:

$$\pi = (s_0, a_0), (s_1, a_1), \ldots$$

Let $\Pi_{States,Actions}$ denote the set of paths over *States* and *Actions*; we will suppress the subscript on Π when *States* and *Actions* are clear from context. In the abstract model, the precise nature of the sets *States* of states and *Actions* of actions is irrelevant; in particular we are not assuming a finite-state framework. In order to represent the fact that obligations are conferred upon, and by, different *agents* in a system, we assume that each action is associated with an agent, or with several agents acting concurrently.

A *system* $\mathcal{S}$ is a set of paths. If $\rho \in \mathcal{S}$ we say that ρ is a *run* of $\mathcal{S}$. In the usual way we can model terminating runs by incorporating a halting state which transitions only to itself.

3.1 Policies

Obligations arise most frequently as an aspect of a rich notion of access control: "permissions with strings attached," in Minsky and Lockman's phrase [4]. Our conception of pure access control is standard: in a given state a policy evaluates a request to perform an action and returns a decision (possible decisions generally include Permit and Deny). We make no assumptions about the language in which policy rules are expressed.

Since our interest in this paper is in access-control policies enriched with a notion of obligations, henceforth the term *policy* will refer the following notion.

Definition 1 A *policy* for a system $\mathcal{S}$ is a tuple $(Decs, Obls, P, \gamma)$, where

– *Decs* is a set of access decisions,
– *Obls* is a set of *obligations*,
– $P : States \times Actions \to Decs \times 2^{Obls}$, and
– $\gamma : Obls \to 2^{\Pi}$.

P determines the access control decision and defines the obligations that arise at each state-action pair. If Ω is an obligation, $\gamma(\Omega)$ is the set of paths which satisfy Ω; we say that this set of paths is the *obligation condition* for Ω. For brevity we will often simply use P to refer to the policy. A system together with a policy determine a *policy-informed program* [5]. This definition does not stipulate how the program handles denials, but is flexible enough to permit the pruning of such paths from the obligation condition.

Consider Example 1 from Section 2. Assume the system maintains information about availability and credentials in relations *Available(f)* and *MayEdit(d,f)*, where f and d denote files and developers, respectively. Let states of the system be sets of facts (closed atomic formulas) over these relations. Assume the system recognizes actions *RequestEdit(d,f)* and *CheckIn(d,f)* to denote that developers want to check out and return files, respectively. The following policy elements formally capture this scenario:

– *Obls* contains one obligation $O_{d,f}$ for each developer d and file f.
– $P(s, RequestEdit(d,f)) = \langle \mathsf{Permit}, \{O_{d,f}\} \rangle$ if *Available(f)* and *MayEdit(d,f)* are both true in s, otherwise $P(s, RequestEdit(d,f)) = \langle \mathsf{Deny}, \emptyset \rangle$.
– $\gamma(O_{d,f}) =$ the set of all paths that contain (at some stage) action *CheckIn(d,f)*.

As obligations specify behavior that a system may or may not respect, we introduce terminology for various relationships between systems and the obligations arising from policies.

Definition 2 If $\pi = (s_0, a_0), (s_1, a_1), \ldots$ is a path and P is a policy, we say that obligation Ω is *created* at stage $i > 0$ of π if $\Omega \in P(s_{i-1}, a_{i-1})$; in this case π *satisfies* Ω if the path $(s_i, a_i), (s_{i+1}, a_{i+1}), \ldots$ is in $\gamma(\Omega)$. The path π *obeys the policy* P if it satisfies each obligation created at each stage of the path. A system obeys a policy P if each of its runs obeys P.

These definitions treat obligations as constraints on entire paths. As such it does not, in general, make sense to speak about them being true or false at a specific state of a computation. But we do note that sometimes it makes intuitive sense to speak of an obligation being discharged at a certain time (e.g., making a log entry) or being violated at a certain time (e.g., publishing a file in violation of a privacy policy). This can be captured formally as follows.

Definition 3 Let $\mathcal{S}$ be a system, let $\rho = (s_0, a_0), (s_1, a_1), \ldots$ be a run of $\mathcal{S}$, and let Ω be an obligation created at stage i of ρ. The obligation Ω is *discharged* at a stage $n \geq i$ of π if every run of $\mathcal{S}$ with prefix $(s_0, a_0), \ldots, (s_n, a_n)$ satisfies Ω. The obligation Ω is *violated* at stage $n \geq i$ of ρ if no run of $\mathcal{S}$ with prefix $(s_0, a_0), \ldots, (s_n, a_n)$ satisfies Ω.

According to the standard taxonomy of system properties defined in Alpern and Schneider [6] an obligation is violable if and only if its condition is a *safety condition*; it is easy to see that an obligation is dischargeable if and only if the negation of its condition is a safety condition; following Manna and Pnueli [7] we refer to these as *guarantee conditions*. (What Manna and Pnueli term an "obligation", however, is merely a boolean combination of safety and guarantee conditions.)

It is often useful to be able to *monitor* a system for policy compliance, especially when dealing with black-box components or those running on untrusted platforms. It is precisely the safety obligations that, in principle, can be the target of runtime monitoring. In Section 5 we consider the problems of detecting whether an obligation is a safety or guarantee obligation, and if not, how to compute an appropriate "best approximation."

3.2 Aren't Obligations Just Fairness Properties?

Our view of obligations as constraining future execution behaviors (paths) of a system suggests a theoretical connection between obligations and the notion of fairness. We cannot, however, reduce obligations to fairness for a few reasons. For one, fairness is a static and global property of a system's execution (usually of the environment), whereas obligations arise dynamically based on actions. This marks obligations as a special class of progress properties. More subtly, fairness is treated as an *assumption* (which may be verified) so that, for instance, a verifier excludes unfair paths; in contrast, while obligations *should* be met, systems expect them to be violated and seek compensation. Video rental stores, for example, oblige customers to return videos by a deadline, but a priori specify late fees because they expect some customers to violate the obligation (and benefit financially from their doing so!).

3.3 Path Specifications

In fact, our formal model has avoided fixing any particular language for paths at all. Instead, it allows any specification of paths for describing conditions; standard languages for paths, such as automata and temporal logic, seem natural concrete choices. If we do choose to use temporal logic, the set of obligations that arise in practice suggest that a minimal set of operators for capturing conditions should include those of LTL, including both the strong and weak variants of the until operator, as well as the unless operator (the dual of until). Example 1 of Section 2 is naturally specified using the "eventually" operator of LTL, while Example 2 clearly requires the "globally" operator.

The application at hand may require that actions of particular agents lead to obligations being fulfilled. See Example 3. LTL is not rich enough to capture the nuances of having several agents, but temporal logics such as ATL* [8] do offer such capabilities. Situations such as Example 4 require something more than propositional temporal logics; this will be explored in detail in the next section,

Finally, many obligations involve bounded time, or intervals of time. Examples include requiring someone to pay by a particular time, or to reverify contact information on an annual basis. While LTL can express such constraints using the next-state operator, such specifications are often clumsy. Other logics [9] provide more nuanced handling of time constraints; the choice of language for describing time is closely tied to to the choice of program model.

The main point of this discussion is to highlight the range of path specification languages that may be useful in devising a concrete language for obligations. We strongly believe that a useful theoretical treatment of obligations should be independent of, or at least parameterized over, the program models and path specifications that arise in particular applications.

4 Obligations Have State

Many interesting notions of obligation go beyond requiring that a single action be taken (or forbidden). This leads us to confront some subtle issues concerning modeling the interaction of policies and programs. Consider Example 4. Suppose A buys an item from B for 10 dollars. Describing the obligation condition as a payment action for 10 dollars isn't right, because debt can be paid in installments. Therefore, tracking the obligation may require maintaining state. In addition to enabling obligations to track state, we should also permit policy authors to use a different vocabulary than that of the program's internal data structures (for instance, the obligation may be in terms of what is owed, while the program only tracks what has been paid).

To make this precise we now settle on some notation for describing program states and the structure of individual obligations. First we tackle the problem of distinct vocabularies. By a *signature* Σ we mean a graded set of relation symbols and constants; we let Facts_Σ denote the set of all closed atomic formulas over such a signature. We assume that states of a system are Herbrand structures over signatures; so states are subsets of the set Facts_{Σ_S} of all facts over signature Σ_S. Other recent works have employed similar models of software as transition systems over relational facts [10, 11].

The policy author works over a different relational signature Σ_O of the terms that capture the state of an obligation. States in policy-informed programs are comprised of facts over $\Sigma_S \cup \Sigma_O$; let $States^{Ob}$ denote the set of all such states.

We can now model the *state* associated with each obligation as follows.

Definition 4 An *obligation state* over Σ_O for a system over *States* and *Actions* is a tuple containing at least two fields: (1) a representation of a set of paths in $\Pi_{States^{Ob},Actions}$ (capturing the condition) and (2) a subset of Facts_O. The set of all such states is denoted *ObStates*.

The "at least" in this definition allows a particular policy language to store additional information with an obligation. Section 6 gives examples of such extensions that arise in practice.

When a policy author defines Σ_O in order to express obligations, she also needs to specify how the obligation state evolves based on system states and actions. This is given in the form of a function.

Definition 5 An *update function* for Σ_O, *States* and *Actions* is a function U of type $States \times Actions \times \text{Facts}_O \to \text{Facts}_O$. U^{Ob} denotes this function lifted to $States \times Actions \times ObStates \to ObStates$ by applying U to each element of Facts_O in an *ObStates*.

Consider Example 4 again. Let *Pay(A,B,n)* be a program action denoting *A* paying *n* dollars to *B*, and *Owes(A,B,x)* denote the internal state of the obligation that records the current debt. The update function is the natural one: most actions leave the balance unaltered, but an action *Pay(A,B,y)* transforms the state of the debt to *Owes(A,B,(x-y))*.

Example 3 regarding delegation also highlights the importance of state in obligations. The state of the obligation associated with a bug would include a field for the agent currently responsible for the obligation. This field can be written by the update function in response to a system action corresponding to delegation of responsibility for the bug. Section 6 discusses why creating separate obligations on each delegation is not necessarily an appropriate alternative to capturing state.

Having refined what it means to be an obligation, we must now refine what it means for a path to satisfy an obligation (thereby refining Definition 2). We first define a function U^* that maps paths over $(States \times Actions)$ to paths over $(States^{Ob} \times Actions)$ as follows.

Definition 6 Let U be an update function, and let β_0 be a subset of Facts_O. When π is a path over $(States \times Actions)$, the path $U^*(\pi, \beta_0)$ is the path whose actions are the same as that of π and whose states are given by

$$\begin{aligned} s_0^* &= (s_0, \beta_0) \\ s_{n+1}^* &= (s_{n+1}, U(s_n, a_n)) \end{aligned}$$

Definition 7 Let Ω be an obligation with condition ϕ and initial set of Σ_O facts β_0. Then Ω is satisfied in a path π over Σ_S if and only if $U^*(\pi, \beta_0)$ satisfies ϕ.

5 Static Analysis and Monitoring of Obligations

We now discuss two styles of analyses, one static and the other dynamic.

5.1 Automata-Theoretic Static Analyses

Having provided a basic model of obligations, we now turn to some analyses we might want to perform.

1. Does a certain run of the system satisfy a given obligation Ω?
2. Does there exist a run of the system in which Ω is satisfied?
3. At a particular state in a run, has obligation Ω been discharged or violated?
4. Does a given model of a policy-informed program satisfy all of its obligations? (This is useful as different models of the same system and its environment may satisfy different obligations.)
5. For a given system condition ϕ, is ϕ satisfied by all runs of the system that satisfy their obligations?
6. When does one obligation imply another in the context of the given system? Are two given obligations contradictory? Does one obligations policy entail another in an absolute sense? Does one obligations policy entail another in the context of the given system?

In this section we assume a finite-state representation of programs and policies. In particular we assume that for a given policy P the set *Obls* of obligations is finite. We further assume that the obligations in question do not involve different agents in any essential way. The generalization of the automata-based techniques [12] of this section to the multi-agent setting is a topic of future work.

In this setting the essential strategy for answering questions such as those above is to capture obligations by automata. In this section we show in Corollary 9 that for each obligation Ω we can construct a Büchi automaton $\mathcal{A}_\Omega$ that accepts precisely those paths that satisfy Ω. In Theorem 10 we show how to combine such automata for the individual obligations to build a Büchi automaton $\mathcal{A}_P$ that accepts precisely those paths that satisfy (all of the obligations in) policy P.

Now suppose that the set of runs of the system $\mathcal{S}$ is given by a Büchi automaton $\mathcal{A}_\mathcal{S}$. If we then take the cross product of $\mathcal{A}_\mathcal{S}$ with $\mathcal{A}_\Omega$ then the resulting automaton accepts precisely the system runs that satisfy Ω. Questions such as 1, 2, and 3 can then be answered using standard algorithms over Büchi automata. If we instead take the cross product of $\mathcal{A}_\mathcal{S}$ with $\mathcal{A}_P$ we can address questions such as 4 and 5. Questions such as 6 are equivalent to questions of language containment for Büchi automata, for which algorithms are well-known [13].

The rest of this section develops these ideas formally.

Definitions A Büchi automaton is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, \Delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial *states*, Σ is a input alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, and $F \subseteq Q$ is the set of *fair states*. A *run* of $\mathcal{A}$ on an infinite word $\alpha \in \Sigma^\omega$ is an infinite sequence r of states from Q such that $r(0) \in Q_0$ and $(r(i), \alpha(i), r(i+1)) \in \Delta$ for all $i \geq 0$. Such a run is *accepting* if $r(i) \in F$ for infinitely many i. The word α is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on α.

Let Φ_O abbreviate $|\text{Facts}_O|$, the number of facts over the signature Σ_O.

Proposition 8. *Let $\mathcal{A}$ be a Büchi automaton over $2^{\mathrm{Facts}_S \cup \mathrm{Facts}_O}$; let β_0 be a set of Σ_O facts. Then there is a Büchi automaton $\mathcal{A}_*$ over 2^{Facts_S} such that for every path π over $(States \times Actions)$*

$$\mathcal{A}_* \textit{ accepts } \pi \textit{ if and only if } \mathcal{A} \textit{ accepts } U^*(\pi, \beta_0)$$

The size of $\mathcal{A}_*$ is $|\mathcal{A}| \cdot 2^{\Phi_O}$.

Proof. This is an instance of a general construction, which will be easier to outline in a more abstract setting.

Let Σ_A and Σ_B be alphabets, and suppose $u : \Sigma_A \longrightarrow \Sigma_B$. Then from any ω-sequence $\alpha = a_0, a_1, \ldots$ over Σ_A and initial element b_0 from Σ_B we can build an ω-sequence $u^*(\alpha, b_0)$ over $\Sigma_A \times \Sigma_B$: $(a_0, b_0), (a_1, b_1) \ldots$ where each b_{i+1} is $u(a_i)$.

It will suffice to show in this setting that given a Büchi automaton $\mathcal{A}$ over $\Sigma_A \times \Sigma_B$ we can build a Büchi automaton $\mathcal{A}_*$ over Σ_A such that for every sequence such as α above,

$$\mathcal{A}_* \text{ accepts } \alpha \text{ if and only if } \mathcal{A} \text{ accepts } u^*(\alpha, b_0)$$

since we may take Σ_A to be 2^{Facts_S}, Σ_B to be 2^{Facts_O} (note that $2^{\mathrm{Facts}_S \cup \mathrm{Facts}_O}$ is naturally isomorphic to $2^{\mathrm{Facts}_S} \times 2^{\mathrm{Facts}_O}$), b_0 to be β_0 and u to be U^*.

Let $\mathcal{A}$ be $((\Sigma_A \times \Sigma_B), Q, Q_0, \Delta, F)$. Define $\mathcal{A}_*$ to be $(\Sigma_A, (Q \times \Sigma_B), (Q_0 \times \{\beta_0\}), \Delta_*, (F \times \Sigma_B))$ where Δ_* is defined by

$$\Delta_*((s,b),a) \;=\; (\Delta(s,(a,b)), u(a))$$

To see that this works, consider a sequence α and let r be any run of $\mathcal{A}_*$ on α, $r = (s_0, b_0), (s_1, b_1), \ldots$. First note that each b_{i+1} is precisely $u(a_i)$. It is then easy to see that the sequence $s_0, s_1, \ldots$ obtained by taking the first component of each pair in r is a run of $\mathcal{A}$ on $u^*(\alpha, b_0)$, and furthermore all such $\mathcal{A}$-runs can be obtained in this way. This establishes the desired relation between $\mathcal{A}_*$ and $\mathcal{A}$.

Corollary 9. *Let Ω be an obligation for system $\mathcal{S}$ whose condition is expressed as a temporal logic formula of size $|\Omega|$. We can build a Büchi automaton $\mathcal{A}_\Omega$ accepting precisely those paths $\pi \in 2^{\Pi}$ that satisfy Ω. The size of $\mathcal{A}_\Omega$ is bounded by $2^{|\Omega| + \Phi_O}$.*

Proof. Let $\mathcal{A}$ be a Büchi automaton over $2^{\mathrm{Facts}_S \cup \mathrm{Facts}_O}$ corresponding to the obligation condition for Ω, and let β_0 be the initial Facts_O facts for Ω. As is well-known, the size of $\mathcal{A}$ is bounded by $2^{|\Omega|}$. The automaton $\mathcal{A}_*$constructed as in Proposition 8 is of size $2^{|\Omega|} \cdot 2^{\Phi_O} = 2^{|\Omega| + \Phi_O}$ and accepts those paths π such that $\mathcal{A}$ accepts $U^*(\pi, \beta_0)$. By Definition 7 these are the paths that satisfy the condition for Ω, so we may take $\mathcal{A}_\Omega$ to be $\mathcal{A}_*$.

Note that the factor of 2^{Φ_O} above is a generous bound: the actual set of Σ_O facts arising in the states of $\mathcal{A}_*$ is the set of facts which can arise in a sequence of U-updates to the initial Σ_O facts in the obligation Ω.

It is not surprising that a given obligation can be represented as an automaton. Each of the infinitely many paths through a system, though, induces its own sequence of obligations to be fulfilled. Hence, it is perhaps surprising that these obligation automata can be combined into a single finite-state automaton for the whole system.

Theorem 10. *Let P be a policy for system $\mathcal{S}$. We can build a Büchi automaton $\mathcal{A}_P$ accepting precisely those paths $\pi \in 2^{\Pi}$ that satisfy P.*

Proof. Given $\mathcal{A}_\Omega$, define the automaton $\mathcal{A}'_\Omega$ which, intuitively, "sleeps" until a system transition is taken that creates the obligation Ω. Formally, we construct $\mathcal{A}'_\Omega$ by starting with $\mathcal{A}_\Omega$ and adding a new state q_0 to $\mathcal{A}_\Omega$, which will serve as the sole initial state. The transition relation Δ' is the extension of the transition function Δ of $\mathcal{A}_\Omega$ defined as follows.

For each pair (s,a):

- Add $\{(q_0,(s,a),q_0)\}$ to Δ';
- if obligation Ω arises due to action a out of s, that is, if $\Omega \in P(s,a)$, then add $\{(q_0,(s,a),r) \mid r \text{ was an initial state of } \mathcal{A}_\Omega\}$ to Δ'.

The automaton $\mathcal{A}_P$ is the product $\prod\{\mathcal{A}'_\Omega \mid \Omega \in \mathit{Obls}\}$.

5.2 Dynamic Monitoring of Safety and Guarantee Obligations

We have seen that a violable obligation is one whose condition is a safety condition, and a dischargeable obligation is one whose condition is a guarantee condition. Under the natural topology on the set $\Pi_{\mathit{States,Actions}}$ of all paths of a system the safety conditions are precisely the closed sets and the liveness conditions are precisely the open sets [6].

The previous notions generalize immediately to the set of runs of a system $\mathcal{S}$: we simply take the subspace topology. In this way we may define the notion of *$\mathcal{S}$-safety condition* and *$\mathcal{S}$-guarantee* condition. A set of paths might define a safety condition (for example) relative to a system even if it fails to be a safety condition in the absolute sense. This yields the appropriate notions of safety and guarantee relevant to obligations being discharged or to be violated in a system. In general, obligations that forbid some action are $\mathcal{S}$-safety conditions, while obligations that demand some action eventually are $\mathcal{S}$-guarantee conditions. Any obligation with a deadline is both $\mathcal{S}$-safety and $\mathcal{S}$-guarantee. Some obligations are neither: the canonical examples are LTL "until" conditions.

Suppose that $\mathcal{S}$ is finite-branching, that is, for every state s there are only finitely many pairs of the form (s,a) that can arise in the system. By an easy application of Kőnig's Lemma, if $\mathcal{P}$ is both an $\mathcal{S}$-safety and an $\mathcal{S}$-guarantee and $\mathcal{S}$ is finite-branching then Ω has a "deadline". That is, there is a single i such that for every run ρ of $\mathcal{S}$, Ω is either discharged or violated before stage i.

Approximating Obligations for Monitoring When a policy cannot be monitored precisely (such as one that is neither safety nor guarantee) and must be approximated, it is valuable to construct a "best" approximation.

Definition 11 Let $\mathcal{P}$ be any condition. The *$\mathcal{S}$-safety closure* $\widehat{\mathcal{P}}$ of $\mathcal{P}$ is the intersection of all $\mathcal{S}$-safety conditions containing $\mathcal{P}$. The *$\mathcal{S}$-interior* $\mathcal{P}^o$ of $\mathcal{P}$ is the union of all $\mathcal{S}$-guarantee conditions contained in $\mathcal{P}$.

The $\mathcal{S}$-safety closure of a condition is a safety condition: it is closed just because closed sets are closed under intersection. Clearly $\widehat{\mathcal{P}}$ is the smallest $\mathcal{S}$-safety condition containing $\mathcal{P}$. Of course, when $\mathcal{P}$ is a $\mathcal{S}$-safety condition, $\widehat{\mathcal{P}} = \mathcal{P}$. Similarly, the $\mathcal{S}$-interior $\mathcal{P}^o$ of a condition is the largest $\mathcal{S}$-guarantee condition contained in $\mathcal{P}$.

Given an arbitrary $\mathcal{P}$, the monitoring system can monitor the $\mathcal{S}$-safety condition $\widehat{\mathcal{P}}$. If a run fails $\widehat{\mathcal{P}}$ it certainly fails $\mathcal{P}$ and, by definition, $\widehat{\mathcal{P}}$ is the smallest condition which can be used to soundly check $\mathcal{P}$-failure. By an analogous argument we may view $\mathcal{P}^o$ as the best dischargeable approximation to $\mathcal{P}$: if a run satisfies $\mathcal{P}^o$ then it is guaranteed to satisfy $\mathcal{P}$, and $\mathcal{P}^o$ is the largest $\mathcal{S}$-guarantee condition with this property.

Computing the Approximations Alpern and Schneider [14] characterized the Büchi automata that accept safety properties. It is easy to see from their analysis how to construct, from a given automaton $\mathcal{A}$, an automaton accepting the closure of the language of $\mathcal{A}$. So to generate an automaton characterizing the $\mathcal{S}$-safety closure of an obligation Ω, construct $\mathcal{A}_\Omega$ as in Section 5.1 and compute its closure. Complementation yields an algorithm for $\mathcal{S}$-guarantee approximation.

6 Evaluation: Modeling Black's

The American legal community has a well-developed taxonomy of obligations in their standard reference, Black's Law Dictionary [3, pages 1104-5]. It is therefore useful to consider how many of Black's obligation types can be captured in the framework we have defined.

Black's describes 30 distinct types of obligations (plus several synonyms and one, correal, that combines other types). Two of these (moral and natural) discuss concepts that lie outside the scope of computing systems. Four (contractual, conventional, obediential, and statutory) capture the source or rationale for introducing an obligation. Some models of programs and their environments could include sufficient information to capture these variations, while others do not. We therefore do not consider these in our analysis, but the data structure for obligations in our model could hold the information needed to distinguish these forms given sufficient program models. The rest fall into 21 classes that have different implications for a model of obligations, including how they arise and evolve. Figure 1 describes these classes and their implications for modeling.

The model that we have presented supports all but two of the Black's classes once obligation states are taken to be arbitrary data structures. Divisible obligations are not supported, as we assume a single obligation condition. Substitute obligations could be encoded, but are not supported naturally. Natural support would require the update function to be able to change the condition, which is beyond the scope of the current model (due to the type of the update function). While defining such an update function is easy, defining the semantics of satisfying obligations in this context is harder. Assume that conditions were expressed as temporal logic formulas; our definition of satisfaction requires each condition to hold from the state at which it was created. If the condition could change before being satisfied, the formula would have to be rewritten to capture a statement roughly corresponding to "the specified formula holds unless the conditions occur to change the condition". One could encode a substitute obligation by writing

Category	**Description**	**Requires**
absolute	condition must be discharged as originally stated, with no modification	update function cannot modify agents or condition
accessory	incidental to another obligation	link obligations in data structure
alternative	obliged may satisfy one of several conditions	data structure with multiple conditions or disjunction in condition formulas
bifactoral, joint, community, solidary	multiple obligated or obligating agents party to a single condition	data structure with multiple obligated or obligating agents
conditional	arises from an event that might not occur	conditions in creation rules
conjunctive	multiple conditions required which may be enforced separately	data structure with multiple conditions; conjunction in formulas insufficient as sub-obligations have separate identity
current	currently enforceable but not past due	recognize when obligation is created
determinate	condition refers to objects by identity	object identity in condition language
divisible	can be divided into parts (without the consent of the parties)	ability to change condition and to associate different subconditions with different obligated agents
heritable	successor of obligated may become liable	ability to update agents
indeterminate	condition refers to objects by attribute	object attributes in condition language
perfect	legally enforceable and binding	"strong" interpretation of condition, capturable in formulas
personal	obligated must personally fulfill obligation	notion of which agents act to fulfill obligations (e.g., as expressed in ATL*)
primary	arises from essential purpose of an action	nothing
primitive	must be satisfied before some others	capture ordering on obligations and refer to order during analysis
pure	enforceable and past due	detecting when obligations are created, satisfied, discharged and violated
secondary	incident to a primary obligation or compensates for other unsatisfied obligation	link obligations in data structure; creation policy checks other obligations' status
several	different conditions required for different agents (obligated or obligating)	data structure groups obligations
simple	arises unconditionally	creation rules without conditions
single	no penalty for non-fulfillment	"weak" interpretation of condition, capturable in formulas
substitute	replaces another, extinguished, obligation	ability to delete obligations and to tie creation to status of other obligations

Fig. 1. Classes of obligations in Black's and their implications for a comprehensive model.

such a condition formula at the outset, but an explicit specification through the update function would be clearer, and we believe preferable.

The Black's classes illustrate that the structure of obligations is both subtle and important. Consider an obligation of the form “do A; otherwise do B and C”. This is a default obligation with a conjunctive obligation in its secondary clause. Because of such conditions, we cannot rely on just the conjunction of top-level obligations to capture all the desired obligation structure. Identity is critical for understanding divisible obligations: the Black's description is ambiguous on whether the component obligations should have separate identities, but whether they do has implications for formal analysis of obligations. In either case, it is clear that our model needs to have a way to refer to an obligation as an identifiable entity, even as its condition potentially evolves. This justifies our model containing both a set of obligations (effectively naming individual obligations) and a separate association of obligation states with obligations.

7 Related Work

Deontic logic [15] is a formal system concerned with reasoning about obligations. Indeed, deontic logic has frequently been used to analyze the structure of normative reasoning in the law. Standard deontic logic is a modal logic, with a unary modality **ob**, so if ϕ is a formula then $(\mathbf{ob}\ \phi)$ is a formula. Formulas are interpreted over Kripke structures, where states represent possible worlds. Permission is precisely the dual of obligation in this logic, so it is natural that several authors ([16–18] and others) have approached the interaction between authorization and obligations in computing systems from this perspective. The approach we pursue here is crucially different from the modal logic approach. Since we view obligations as expressing constraints on computations, it does not make sense to ask whether an obligation “holds” at a state.

Minsky and Lockman recognized the essential association of obligations with permissions some time ago [4]. Their informal syntax and semantics supports a rich taxonomy of obligations (including deadlines and both positive and negative obligations), but does not handle state. Mont's [19] rich taxonomy of privacy obligations for enterprises provides more detailed implementation requirements and state contents for obligations than ours, but lacks formal semantics and our theoretical treatment of analysis. Mont's model supports some features, such as compensatory actions and additional requirements on future actions, that are not cleanly expressible in our policy notations.

Irwin, Yu, and Winsborough [20] propose a formal model of obligations inspired by the idea that an obligation is a contract between a system and a subject. They define a notion of secure system state based the concept of *accountability* for violation of an obligation, and explore the complexity of checking accountability properties. Though there are many similarities between their approach and ours, we have a somewhat more general system model and a richer semantics of obligations.

Several works focus mainly on specifying, rather than analyzing, obligations. Park and Sandhu [21] view obligations as current or past conditions constraining access requests (resembling provisions [22]). Constraints on future behavior are limited to a predicate that must hold so long as the system retains access to an object. Our model divorces obligations from the lifetime of permissions, and can also associate them with denied requests. Sloman [23] distinguishes authorization (actions that may occur) from obligation (actions that must or must not occur). The Ponder policy language [24] views obligations as actions that must be executed when certain events occur, but these conditions do not support temporal operators. Kudo and Hada [25] fix a set of primitive obligations (such as logging) that happen when access is granted, but also fail to support temporal operators.

Some models support analysis of obligations policies outside the context of program models. Abrahams, et al.'s [26] model allows for obligations to not be satisfied, and can associate obligations with denials. Temporal constructs on obligations are implicit and limited to eventuality (for positive obligations) and globally (for negative obligations). Schaad and Moffett [27–29] explore interactions between agents in the context of obligations, addressing delegation, review and supervision, and revocation. They represent and analyze obligations using Alloy [30] but do not present a general formal semantics for them. Bettini, et al. [22] model access-control policies with both provisions and obligations. Their model, like ours, links obligations to policy rules and defines them using a separate signature of terms. It includes actions to be executed when an obligation is satisfied or fulfilled. They focus on a semantics for when obligations apply, but assume that atomic obligations are interpreted by a server, which they do not model. Backes, et al. [31] extend EPAL with a model of obligations and containment between them, and also assume a server for atomic obligations. This assumption makes monitoring possible at the expense of less expressive obligations (Section 5.2). Neither work accounts for the stateful nature of obligations, which this paper shows is extremely valuable for modeling numerous scenarios (and complicates the definition of obligations).

Work that explores obligations in the context of program models tends to focus on the impact of programs on obligations, whereas our more general model supports reasoning about the impact of obligations on programs. These works also fail to capture obligations with state. Hilty, Basin and Pretschner [32] explore the role of obligations for data providers, focusing on privacy and intellectual property management. They specify obligations using Distributed Temporal Logic (DTL) [33], which supports a rich notion of agents. They discuss a variety of strategies for enforcing non-monitorable obligations by weakening them; we have outlined a systematic approach for approximating obligations motivated by enforcement. May, et al. [34] model privacy policies with obligations using an extension of the classical access-control matrix. Their work explores the interactions of policies and programs, but not the implications of this interaction on the structure of obligations (as in our Section 4). Barth, et al. [35] model privacy policies as rules conditioned on both past and future behaviors and programs as sequences of events that transmit information. Unlike us, they assume agents cannot violate obligations.

8 Conclusion and Future Work

This paper has presented a model of obligations and their interaction with an ambient system. Obligations are viewed as a means for expressing constraints on the future behavior of a system, have state, and can fail to be fulfilled. The combination of these assumptions, the generality of our model, and our model of system-policy interaction distinguish our work from other treatments of obligations. The paper has demonstrated several useful analyses including a systematic means for approximating obligations for monitoring.

Although the paper has demonstrated that this model of obligations is quite rich, it cannot support some kinds of obligations that arise in practice, such as divisible and substitute obligations (see Section 6), and obligations that require new behavior (such as additional authentication checks) on every subsequent execution of a particular action [19]. The latter reflects a fundamental assumption in our model that the policy does not add, remove, or alter behaviors of the system. Relaxing this assumption is an important topic for future work.

Next, because obligations may not be fulfilled, it is natural to speak of compensatory actions (akin to those used in database transactions). While these can be encoded in the obligation conditions in our model, they enjoy no distinguished status and thus cannot be reasoned about directly. It could be useful to ask, for example, whether a system that fails some property in the presence of unfulfilled obligations will satisfy the property under a specific set of compensations, or to try to synthesize information about the compensations that could cover all unfulfilled obligations.

Further, the ability to decide various questions about obligations using automata theory points the way to a *logic of obligations*. This would offer a contrast to attempts to apply deontic logic to obligations.

Finally, we would like to support richer notions of agents and analyses that account for them. Section 3.3 raised the question of whether a particular agent could fulfill its obligations. It would also be interesting to try synthesizing minimal models of agents that guarantee satisfaction of their obligations. Both questions require closer attention to agent-aware logics like ATL* for specifying conditions and their corresponding models of system behavior.

Acknowledgments We thank Dan Ignatoff and Felix Geller for their assistance on this project, and the anonymous reviewers for their helpful comments. This work is partially funded by NSF grant CNS-0627310. We commemorate the passing of Cupcake during the authoring of this paper.

References

1. Moses, T.: eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS (February 2003)
2. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise privacy authorization language (EPAL 1.2). `http://www.w3.org/Submission/EPAL/`
3. Garner, B.A., ed.: Black's Law Dictionary, 8th edition. Thomson-West Publishers (2004)
4. Minsky, N.H., Lockman, A.: Ensuring integrity by adding obligations to privileges. In: International Conference on Software Engineering. (1985) 92–102
5. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: International Joint Conference on Automated Reasoning. (August 2006) 632–646
6. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21** (1985) 181–185

7. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York (1992)
8. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM **49**(5) (2002) 672–713
9. Alur, R., Henzinger, T.A.: Logics and Models of Real-Time: A Survey. In: Real Time: Theory in Practice. (1991) 74–106
10. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web services. In: Principles of Database Systems. (2004) 71–82
11. Yahav, E., Reps, T., Sagiv, M., Wilhelm, R.: Verifying temporal heap properties specified via evolution logic. In: European Symposium on Programming. (2003) 204–222
12. Vardi, M.Y.: Verification of concurrent programs: The automata-theoretic framework. Annals of Pure and Applied Logic **51**(1-2) (1991) 79–98
13. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
14. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing **2**(3) (1987) 117–126
15. von Wright, G.H.: Deontic logic. Mind **60** (1951) 1–15
16. Bartha, P.: Conditional obligation, deontic paradoxes, and the logic of agency. Annals of Mathematics and Artificial Intelligence **9**(1-2) (1993) 1–23
17. Jamroga, W., van der Hoek, W., Wooldridge, M.: On obligations and abilities. In: Deontic Logic in Computer Science. (2004) 165–181
18. Kooi, B.P., Tamminga, A.M.: Conflicting obligations in multi-agent deontic logic. In: Deontic Logic in Computer Science. (2006) 175–186
19. Mont, M.C.: A system to handle privacy obligations in enterprises. Technical Report HPL-2005-180, HP Laboratories Bristol (October 2005)
20. Irwin, K., Yu, T., Winsborough, W.H.: On the modeling and analysis of obligations. In: Computer and Communications Security. (2006) 134–143
21. Park, J., Sandhu, R.: The UCON$_{\mathrm{ABC}}$ usage control model. ACM Transactions on Information and System Security **7**(1) (February 2004) 128–174
22. Bettini, C., Jajodia, S., Wang, X., Wijesekera, D.: Obligation monitoring in policy management. In: Policies for Distributed Systems and Networks. (2002) 2–12
23. Sloman, M.: Policy driven management for distributed systems. Journal of Network and Systems Management **2**(4) (1994)
24. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder specification language. In: Policies for Distributed Systems and Networks. (2001)
25. Kudo, M., Hada, S.: XML document security based on provisional authorization. In: Computer and Communications Security. (2000) 87–96
26. Abrahams, A., Eyers, D., Bacon, J.: An asynchronous rule-based approach for business process automation using obligations. In: Rule-Based Programming. (2002) 93–104
27. Schaad, A., Moffett, J.D.: Delegation of obligations. In: Policies for Distributed Systems and Networks. (2002) 25–35
28. Schaad, A.: An extended analysis of delegating obligations. In: Data and Applications Security. (2004) 49–64
29. Schaad, A.: Revocation of obligation and authorisation policy objects. In: Data and Applications Security. (2005) 28–39
30. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology **11**(2) (April 2002) 256–290
31. Backes, M., Pfitzmann, B., Schunter, M.: A toolkit for managing enterprise privacy policies. In: European Symposium on Research in Computer Security. (2003) 101–119
32. Hilty, M., Basin, D.A., Pretschner, A.: On obligations. In: European Symposium on Research in Computer Security. (2005) 98–117

33. Ehrich, H.D., Caleiro, C.: Specifying communication in distributed information systems. Acta Informatica **36**(8) (2000) 591–616
34. May, M.J., Gunter, C.A., Lee, I.: Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In: Computer Security Foundations Workshop. (2006)
35. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: Framework and applications. In: Symposium on Security and Privacy. (2006)