The Specification and Compilation of Obligation Policies for Program Monitoring

Cheng Xu and Philip W. L. Fong
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{ cxu, pwlfong }@ucalgary.ca

April 14, 2011

Abstract

The core component of an extensible software system must protect its resources from being abused by untrusted software extensions. The access control policies of extensible software systems are traditionally enforced by some form of reference monitors. Recent study of access control policies advocates the use of obligation policies, which impose behavioural constraints to the future actions of the accessor even after the access is granted. It is argued that obligation policies provide continuous protection to the system.

We envision the workflow of developing an obligation policy for program monitoring to involve three stages: specification, implementability check and implementation. In this work, we develop a toolchain to support each stage of the workflow. First, we propose a policy language for formulating obligation policies. Second, we devise a type system for syntactically identifying if an obligation policy is enforceable or not. The type checker guides the policy developer in refining an obligation policy into an enforceable one. Finally, we design a compilation algorithm, which compiles well-typed obligation policies to a representation of reference monitors, called Obligation Monitor (OM). The OM is designed to facilitate monitor inlining.

1 Introduction

1.1 Motivation

Complex software systems are usually structured as extensible systems. An extensible system is composed of a core system and a set of third-party extensions. The latter augments the core's functionality. For example, a mobile operating system, like Android, can be extended by installing other programs; word processors, like Microsoft Word$^{\text{TM}}$, support macro programming; and Internet web browsers, like Firefox, can be extended with plug-ins and webpage applets.

A major threat to extensible systems, like those above, is the lack of assurance about the extensions' behavior. A malevolent extension has the potential to compromise the core system because the two typically share the same set of resources, and in some cases the same address space.

One way to address the security challenge is to deploy a *reference monitor* [4] to ensure that the behavior of the extensions comply with pre-defined security policies. Typically, a reference monitor intercepts all actions of the extensions and, based on the pre-defined policy, takes remedial action on operations that might cause damage to the integrity of the system (e.g. by suppressing the

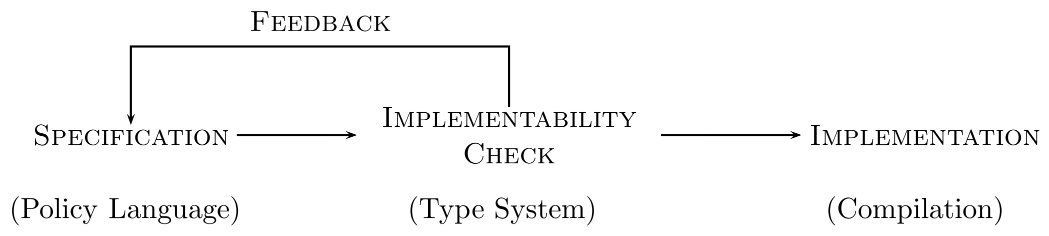

Figure 1: Workflow of developing obligation policies for program monitoring.

offending actions or terminating the extension itself). A modern implementation of this enforcement mechanism is *Inlined Reference Monitor (IRM)* [18], in which, the *target program* (extension) is modified through a trusted program rewriter to include the functionality of a reference monitor. Specifically, the trusted rewriter takes in a pre-defined security policy, compiles it to some security code fragments, and injects them into the target program. These security code fragments intersperse test with the instructions of the target program, and terminate its execution when the program is about to violate the policy.

Previous works [15, 24] noted that some security properties that need to be imposed on untrusted programs are ***obligation policies*** [28, 11]. While a traditional access control policy either grants or denies access, an obligation policy grants access "with strings attached": access is granted with the condition that future actions of the accessor conform to some behavioral constraints. For example, consider the *Logging-Following-Access-Denial* policy [15]: 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.

Compared with access control policies, which aim to restrict the accessors only by providing "yes/no" decisions on access requests, obligation policies prescribe the future constraints on the accessors upon the system's granting of access. Therefore, this coupled with the using of IRMs, can provide continuous protection to extensible system.

1.2 Challenge

The introduction of obligation policies to IRMs is not as straightforward as it looks like at the first glance.

The most important challenge would be that not all obligation policies are enforceable by a reference monitor. Schneider [43] showed that reference monitors that respond to potential security violations by halting the target program can only enforce safety properties — security policies specifying that "nothing bad ever happens" [2] — in a valid run of the target program. Yet, Daugherty *et al.* [15] pointed out that obligation policies relevant to software security are safety and guarantee properties as well as their boolean combinations [13].

Based on this observation, we envision the workflow of developing obligation policies for program monitoring to involve three stages (Figure 1):

1. The ***specification*** stage involves the translation of an informal obligation policies (in natural language) to a formal one.

2. The policy developer then performs an ***implementability check*** to determine if the obligation policy can be enforced by a reference monitor. The result of the implementability check will be fed back to the policy developer, who can either go back to the specification stage, or move forward to the implementation stage.

3. The ***implementation*** stage involves the construction of an enforcement mechanism for an enforceable obligation policy.

Although there are researches related to each stage of the workflow, a unified framework, which provides supports to all three stages, is still missing. Therefore, the goal of this report is to develop a toolchain, which streamlines the workflow as a whole.

1.3 Contributions

The four contributions of this report provide support to the various stages of the workflow:

- First, we propose a policy language for expressing software-based obligation policies. Inspired by [24], formal syntax and semantics for the language has been defined based on temporal logic. We show that this language captures the common idioms of obligation policies in a modular and composable manner. In addition, a number of case studies are presented to demonstrate the use of language in formalizing software-based obligation policies.

- Second, We present a type system for determine if an obligation policy in our policy language is either a safety or a guarantee property [13]. The type system captures the idiomatic compositions of obligation formulae that are enforceable. It also provides feedbacks to policy developers regarding which part of a formula has to be refined in order to make the policy enforceable. We demonstrate through a case study that the type system helps the policy developers in formulating an enforceable obligation policy.

- To facilitate IRM, we design a representation of reference monitor, called ***obligation monitor (OM)***, which enforces safety properties and monitors guarantee properties. We also prove that the OM is as expressive as finite-branching security automata in policy enforcement.

- Finally, We develop a compilation algorithm for compiling well-typed obligation formulae to their corresponding enforcement mechanisms. The compilation algorithm employs the type system to guide its compilation, converting a well-typed formula to an OM. A correctness theorem (which also entails the soundness of the type system) has been proved.

This report follows the progression described by the workflow (Figure 1) by first defining a policy language for specifying obligation policies, followed by introducing the type system that checks the enforceability of an obligation policies (written in the policy language above) and provides feedbacks to the policy developers, and finishing with a compilation algorithm that compiles a well-typed obligation policy to its corresponding reference monitor. Specifically, this report is organized as follows.

Section 2 introduces some basic notations, such as traces, prefix and suffix, safety and guarantee property, and action assertion language. Section 3 presents an obligation policy language with formal syntax and semantics. It also presents case studies to demonstrate the use of the language. Section 4 introduces the type system. A case study is also reported in this section, which demonstrates the use of type system as the feedback provider in formulating enforceable properties. Section 5 presents the model of obligation monitor (OM), and explains the expressiveness and advantages of having such a representation of reference monitor. Section 8 describes the compilation algorithm, which compiles the obligation policies to OMs. Section 9 briefly reviews related works. Finally, in Section 10, we conclude the report and outline some future works.

2 Preliminary

In this section, some basic notations that will be used in this report will be defined. We begin in Section 2.3 by defining the program model used in this report. Then, Section 2.2 provides a formal definition of safety, guarantee, k-bounded safety and k-bounded guarantee property, based on the program model defined in the previous section. Section 2.3 reports a comparison between our program model and the standard program model. During the comparison, we highlighted a small, yet important contribution of this report - a tighter bound for enforceability of properties can be defined by further constraining the capability of an observer in detecting the termination of executions. Lastly, we define in Section 2.4 the action assertion language, which provides the atomic building blocks for the policy language that will be introduced in the next section.

2.1 Notation

Basic Notations. Given a set S, $[S]^k$ denotes the set of all k-element subsets of S, and $[S]^{<\omega}$ denotes the set of all finite subsets of S. The notation 2^S denotes the powerset of S (i.e. the set of all subset of S).

Program Model. At run time, a program performs actions that belong to a set Σ. A program execution can be characterized as a ***trace***, which is either a finite or infinite sequence of actions. Intuitively, a finite trace indicates either a terminated execution or a partial execution. An infinite trace represents a non-terminating execution. We denote by Σ^* the set of finite traces over Σ, and by Σ^ω the set of infinite traces over Σ. Then, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ is the set of all traces over Σ. The empty sequence is denoted by ϵ. Given $u \in \Sigma^*$ and $v \in \Sigma^\infty$, we write uv to denote the concatenation of u and v. Then the notions of ***prefix*** and ***suffix*** can be defined as follows.

Definition 1. (Prefix and Suffix) *Given $u \in \Sigma^*$ and $v \in \Sigma^\infty$, we write $u \cdot v$ (or simply uv) to denote the concatenation of u and v. If $w = uv$, we say u is a* **prefix** *of w, v is a* **suffix** *of w, and we write $u \preceq w$ and $v \sqsubseteq w$. Specifically, if $u \neq w$, then we write $u \prec w$; if $v \neq w$, then we write $v \sqsubset w$. We also write w/v to denote u, and $w - u$ to denote v.*

Note that a prefix is always finite, and a suffix can be either finite or infinite. Moreover, ϵ is a prefix of any trace.

2.2 Safety and Guarantee Properties

A *property* Φ is a subset of traces Σ^∞. Consider a given trace $w \in \Sigma^\infty$, when $w \in \Phi$, we say that w *satisfies* Φ. Note that the membership of w determined by w alone, not by other traces [1]. A property is said to be ***violated*** by a finite trace u if there is no possible extension uv for which uv satisfies the property. A property is said to be ***fulfilled*** by a finite trace u if for every possible extension uv, the property is satisfied by uv.

Dougherty *et al.* [15] identified obligation policies that can be enforced by a reference monitor to be safety properties, and obligation policies for which fulfillment can be detected by a reference monitor to be guarantee properties. Informally,

- a ***safety property*** is a security policy that states some good thing always happens;

[1] This is the distinction, made by Schneider [43], between properties and general policies. The set of properties (defined over single execution sequences) is a subset of the set of policies (defined over sets of execution sequences).

- a ***guarantee property*** [2] is a security policy that states some good thing happens at least once in finite time.

More formally, the two can be defined as follows.

Definition 2. (Safety and Guarantee Property) *A property Φ is*

- *a* **safety property** *if and only if for every $w \in \Sigma^\infty$, $w \notin \Phi$ implies that there exists $u \preceq w$, $u \notin \Phi$ and for any $v \in \Sigma^\infty$, $uv \notin \Phi$;*
- *a* ***guarantee property*** *if and only if for every $w \in \Sigma^\infty$, $w \in \Phi$ implies that there exists $u \preceq w$, $u \in \Phi$ and for any $v \in \Sigma^\infty$, $uv \in \Phi$.*

Note that the dissatisfaction of a safety property implies violation, and the satisfaction of a guarantee property implies fulfillment.

Suppose the violation (respectively fulfillment) of a safety (respectively guarantee) property always occurs within a fixed number $k \in \mathbb{N}$ of steps, such a safety (or guarantee) property is said to be bounded by a finite bound k. More formally,

Definition 3. (k-bounded safety and k-bounded guarantee property) *Let $k \in \mathbb{N}$. A property Φ is*

- *a* **k-bounded safety property** *if and only if it satisfies:*
  1. *Φ is a safety property.*
  2. *for any $u \in \Sigma^*$, $|u| \geq k$ and $u \in \Phi$ implies that, for any $v \in \Sigma^\infty$, $uv \in \Phi$.*
- *a* **k-bounded guarantee property** *if and only if*
  1. *Φ is a guarantee property.*
  2. *for any $u \in \Sigma^*$, $|u| \geq k$ and $u \notin \Phi$ implies that, for any $v \in \Sigma^\infty$, $uv \notin \Phi$.*

If a safety property is a k-bounded safety property, a reference monitor can safety give up detecting violation after failing to find one in the first k steps. Similarly, if a safety property is a k-bounded guarantee property, a reference monitor can safety give up monitoring fulfillment after failing to find one in the first k steps.

2.3 Standard vs. Non-standard Program Model

In this report, we model program executions to be either finite or infinite traces, where a finite trace can be a terminated or a partial execution (i.e. a prefix of a longer trace). Such a way of modeling is slightly different from the standard one in the literature [2, 3, 43, 35, 33, 32, 34] in the following way. The standard model involves only infinite traces (i.e. terminated executions are modeled by infinitely repeating the last symbol `t` — the termination of executions).

Such a shift in modeling program executions indeed means: an observer (e.g. a reference monitor) cannot differentiate a partial execution from a terminated one. This, in fact, allows us to define a tighter bound for enforceable properties.

In the standard program model, in which program executions are modeled as infinite sequences, there is no difference between a k-bounded safety property and a k-bounded guarantee property. For example, a k-bounded safety property Φ is also a guarantee property, because given an infinite

[2] In [10], guarantee properties are named co-safety properties.

trace w with $w \in \Phi$, there exists $u \prec w$, $|u| \geq k$ and for any infinite trace v, $uv \in \Phi$. Yet, in our model, in which partial executions and terminated executions are modeled as finite traces, the two notions (k-bounded safety property and k-bounded guarantee property) refer to distinct families of properties.

Such a way of modeling is also reasonable. Consider a typical reference monitor, which prevents a target program from violating a policy by revoking the offending actions (e.g., denying an access). The action of termination is not revocable. That is why, to a reference monitor, it is not an action that belongs to the set Σ of revocable actions. Allowing the reference monitor to detect the termination of executions should not give it more power in enforcing properties.

2.4 Action Assertion Language

We postulate that an ***action assertion language*** has been provided for classifying actions.

Definition 4. (Action Assertion Language) *Given the set Σ of actions, an action assertion language $\mathcal{L}_\Sigma$ is a pair $\langle \mathcal{P}, \Vdash \rangle$, where*

- *$\mathcal{P}$ is a countable set of* **action propositions***, and,*
- *$\Vdash \subseteq \Sigma \times \mathcal{P}$ is a binary relation specifying which proposition is satisfied by which action.*

The action assertion language provides atomic propositions for our policy language. It can be extended with boolean connectives as follows.

Definition 5. (Action Formulae) *The syntax of an action formula is given below:*

$$\psi ::= \texttt{true} \mid \texttt{false} \mid p \mid \texttt{!}\,\psi \mid \psi\,\texttt{\&\&}\,\psi \mid \psi\,\texttt{||}\,\psi$$

where p is an action proposition.

Such an action formula is merely a boolean combination of action propositions. Let $\Phi(\mathcal{L}_\Sigma)$ be the set of action formulae defined over $\mathcal{L}_\Sigma$. The satisfaction relation ($\Vdash$) can be extended to action formulae in a standard way: let us write $a \Vdash \psi$ whenever action a satisfies action formula ψ.

Note that the ontology of program actions depends on many factors, such as the architecture of target programs (e.g., in a transactional system, the whole transaction is considered as one action), the security policy to enforce (e.g., for usage control policies, only usage of resources are counted as program actions), etc. Defining a unified structure of action proposition is beyond the scope of this report. Generally, the information that can be captured by action propositions are as follows.

- action types (e.g. I/O operations)
- action argument types (e.g. the function call `open(String fileName)` over files with extension `.psd`)
- subject types (e.g. process identifiers)
- other contextual information (e.g. system time)

3 Obligation Policy Language

The language we propose in this section is a policy language for specifying obligations in security policies for software system. It is designed to capture the common idioms of obligation policies in a modular and composable manner. We formalize both the syntax and semantics for the language based on temporal logic.

This section is structured as follows. In Section 3.1, we describe what idiomatic components obligation policies have, i.e., what the policy language is designed to describe. Then, we presented the formal syntax and semantics of the language in Section 3.2. In Sections 3.3 and 3.4, we further define some derived forms for the language, and also identify a set of semantic equivalences in the language. Finally, in Section 3.5, we demonstrate the use of the language in expressing obligation policies through case studies.

3.1 Requirements

To design a language specifying obligation policies, a first question we have to answer is: what is an obligation policy? In many application domains, an obligation policy always specifies the future behaviors that the target application is required to perform or prohibited to perform, in order for the system to allow certain actions to be taken by the application at present. For example, if an application is allowed to reserve certain critical resources from the system, it is obliged to use and release the resources within a number of computation steps.

In this section, we examine some existing structures of obligation policies proposed in other literatures. From those structures, we distill a set of components that are contained in typical obligation policies, so that later in this section, we can show that our language provides syntactic support for expressing each component. Note that here, we are not trying to define a precise model for obligation policies, but only to set up goals for what we hope the language can express.

- Minsky and Lockman [37] proposed a unified policy structure to integrate obligations and permissions as follows:

$$\begin{aligned} \pi \ &::= \ \textbf{can}\ \langle \textit{op-set} \rangle\ \textbf{requiring}\ \langle \textit{obligation} \rangle \\ \langle \textit{obligation} \rangle \ &::= \ \langle \textit{requirement} \rangle\ \textbf{by}\ \langle \textit{deadline} \rangle\ \textbf{or-else}\ \langle \textit{sanction} \rangle \end{aligned}$$

  The policy works during program executions in the following way: whenever an application performs any operation defined by $\langle \textit{op-set} \rangle$, the obligation defined by $\langle \textit{obligation} \rangle$ is triggered, requiring the application to fulfill some requirements specified in $\langle \textit{requirement} \rangle$ by the deadline specified in $\langle \textit{deadline} \rangle$. If such an obligation is not fulfilled, actions defined by $\langle \textit{sanction} \rangle$ are performed by the program monitor to penalize the application.

  Moreover, they further divided the requirements defined by $\langle \textit{requirement} \rangle$ into three types: to do something, to maintain certain invariant, and not to do something.

- In Ponder [14], Damianou *et al.* defined obligation policies as event-triggered rules as shown in Figure 2. Such an obligation policy is triggered by events specified in the `on`-clause. As a result, the `subject` must `do` something on some `target` under certain temporal constraint that is defined by the `when`-clause. The optional `catch`-clause specifies an exception that is executed if the actions fail to execute for some reason.

- Gama and Ferreira [22] thought of an obligation policy as a set of rules in the form $\langle \textit{trigger expression} :: \textit{obliged expression} \rangle$. Each rule states that the obliged expression has to be enforced whenever the trigger expression holds.

```
inst oblig policyName  "{"
     on                                      event-specification ;
     subject                        [⟨type⟩] domain-Scope-Expression ;
     [ target                       [⟨type⟩] domain-Scope-Expression ; ]
     do                                      obligation-action-list
     [ catch                                 exception-specification ;]
     [ when                                  constraint-Expression ;]  "}"
```

Figure 2: Syntax of obligation policies in Ponder

- Irwin *et al.* [28] proposed another obligation policy model, where the notion of obligation policy differs from the notion of obligation. In general, an obligation policy consists of an action, some conditions for performing the action, and a set of obligations resulting from the action. An obligation is formalized as $\langle s, a, O, [t_s, t_e]\rangle$, which states that the subject s is required to perform certain action a over a set O of objects within a time frame $[t_s, t_e]$.

- Hilty *et al.* [24] also differentiated obligations from obligation policies, or in their term, obligations and obligational formulae. According to them, obligation policy is typically a phrase of the form "if *activation*, then *obligational formulae*". The *obligational formulae* becomes an *obligation* after the *activation* of obligation.

  They also introduced in their technical report [25] the concept of compensation as a complement to obligation policies, which specifies the actions to be performed if certain obligation has been violated.

By synthesizing the above structures, a set of components that forms a typical obligation policy are outlined as follows:

Trigger Trigger defines the applicability of an obligation policy, which once triggered, will result in some obligations to be imposed. Examples are: $\langle$*op-set*$\rangle$ in Minsky and Lockman's policy structure, the `on`-clause in Ponder policy, the *trigger expression* in Gama *et al.*'s model, and the notion of *activation* in Hilty *et al.*'s phrase.

Obligation Obligation is the core of an obligation policy. It specifies the actions that are obliged (or prohibited) to be performed by the target program. For example, the $\langle$*requirement*$\rangle$ in Minsky and Lockman's structure, the `do`-clause in Ponder policy, and the $\langle$*obliged expression*$\rangle$ in Gama *et al.*'s model.

Temporal Constraint Temporal constraint defines the time window in which the obligation is imposed. For example, the $\langle$*deadline*$\rangle$ in Minsky and Lockman's structure, $[t_s, t_e]$ in Gama *et al.*'s model, or the notion of *conditions* in Hilty *et al.*'s model.

Note that temporal constraint does not have to explicitly specify a starting instant and a deadline to define such a time window. In fact, the time window can be defined by any form of temporal requirements over program behaviors. For example, the policy with CARDINALITY CONDITIONS [24] "a sensitive file can only be accessed once" can be rephrased as "after the first access to the sensitive file, any future access to that file is not allowed", which also gives us a time window from the starting instant ("the first access to the sensitive file") to infinity in the future.

Penalty (Reward) Penalty (Reward) is the access control implication if the target program violates (or fulfills) the obligation under the temporal constraint.

Penalty (Reward) are meaningful in defining an obligation policy. Intuitively, penalty defines the exception handling for violating the obligation, while reward inspires the fulfilling of the obligation.

We ignore some information that are also shared by some of the policy structures mentioned above, such as subjects and objects. This is because, these information are more related to single program actions, rather than program executions (to which we indeed evaluate the obligation policies).

Except for the obligation component, every component outlined above is optional. Consequently, rather than designing a monolithic syntax for a policy clause, we instead design a reusable construct for each component, and use a temporal logic to provide means to mix and match the components.

3.2 Syntax and Semantics

Our policy language is a temporal logic, in which a formula is evaluated against a trace. Consequently, formulae in the policy language are called trace formulae (ϕ) to distinguish them from action formulae (ψ).

Definition 6. (Syntax) *The syntax of trace formulae (ϕ) is defined in BNF as follows.*

$$\begin{aligned}\phi \quad ::= \quad & \top \mid \psi \mid \neg\phi \mid (\phi \wedge \phi) \mid (\mathsf{Eventually}\ \phi) \mid (\mathsf{Before}^{+}\ \phi : \phi) \mid \\ & (\mathsf{After}^{+}\ \phi : \phi) \mid (\mathsf{Ignoring}\ \psi : \phi)\end{aligned} \tag{1}$$

The operators (e.g. $\neg$, $\wedge$, Eventually, etc.) used in (1) are called ***trace operators***, which only modify trace formulae. Thus, $(!\ \neg p)$ is not a valid trace formula.

Definition 7. (Semantics) *Given a trace $w \in \Sigma^{\infty}$ and a trace formula ϕ, we say w **satisfies** ϕ (written as $w \models \phi$) if and only if:*

1. $w \models \top$
2. $w \models \psi$ *if and only if there exists an action $a \preceq w$ such that $a \Vdash \psi$;*
3. $w \models \neg\phi$ *if and only if* $w \not\models \phi$;
4. $w \models \phi_1 \wedge \phi_2$ *if and only if* $w \models \phi_1$ *and* $w \models \phi_2$;
5. $w \models \mathsf{Eventually}\ \phi$ *if and only if there exists $v \sqsubseteq w$ such that $v \models \phi$;*
6. $w \models \mathsf{Before}^{+}\ \phi_1 : \phi_2$ *if and only if either (a) there exists $u \preceq w$ such that $u \models \phi_1$, for all $u' \prec u$, $u' \not\models \phi_1$, and $u \models \phi_2$, or (b) for all $u \preceq w$, $u \not\models \phi_1$, and $w \models \phi_2$;*
7. $w \models \mathsf{After}^{+}\ \phi_1 : \phi_2$ *if and only if either (a) there exists $u \preceq w$ such that $u \models \phi_1$, for all $u' \prec u$, $u' \not\models \phi_1$, and $w - u \models \phi_2$, or (b) for all $u \preceq w$, $u \not\models \phi_1$*
8. $w \models \mathsf{Ignoring}\ \psi : \phi_2$ *if and only if $w|_{\psi} \models \phi_2$, where $w|_{\psi}$ is the subsequence obtained by removing from w all occurrences of the actions a such that $a \Vdash \psi$.*

Clause 1 indicates that $\top$ is satisfied by any trace. Clause 2 indicates that, the trace formula ψ is satisfied by a non-empty trace with its first action satisfying the action formula ψ. Note that the trace formula ψ specifies a guarantee property.

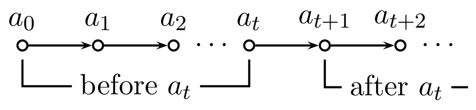

Figure 3: Illustration of "before" and "after" at a certain instant t.

Let's now look at clauses 3 - 5, action operators $\neg$ and $\wedge$ are merely boolean negation and conjunction respectively. Eventually ϕ holds on w whenever ϕ holds for some suffix of w. Eventually is therefore semantically equivalent to the temporal operator $\mathcal{F}$ in LTL [39].

The interpretation of clauses 6 and 7 are illustrated by Figure 3. Consider $\text{Before}^+\ \phi_1 : \phi_2$. Suppose t is the first instant when ϕ_1 is fulfilled by a prefix $a_1 a_2 \cdots a_t$ of w. Then ϕ_2 is imposed on that prefix. If ϕ_1 is never fulfilled, then ϕ_2 is imposed over w itself. The intended usage is that ϕ_1 shall be a guarantee property and ϕ_2 shall be a safety property. When used this way, $\text{Before}^+\ \phi_1 : \phi_2$ is a safety property. Similarly, consider $\text{After}^+\ \phi_1 : \phi_2$. Suppose t is the first instant when ϕ_1 is fulfilled by a prefix $a_1 a_2 \cdots a_t$ of w. Then ϕ_1 is imposed on the suffix $a_{t+1} a_{t+2} \cdots$ of w. If ϕ_1 is never fulfilled, which can be concerned as the first fulfillment of ϕ_1 occurs in the infinite future, there is no such suffix to impose ϕ_2. As well, the intended usage of After^+ is that ϕ_1 shall be a guarantee property and ϕ_2 shall be a safety property. When used this way, $\text{After}^+\ \phi_1 : \phi_2$ is a safety property.

Finally, clause 8 captures the scenario that, by ignoring some actions in the trace, certain formula is satisfied.

3.3 Derived Forms

A set of derived forms can be defined for our language. First, some derived forms are merely standard duals as follows.

$$\bot \stackrel{\text{def}}{=} \neg\top, \qquad \phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2) \text{Always}\ \phi \stackrel{\text{def}}{=} \neg\text{Eventually}\ \neg\phi$$

Trace operators $\bot$ and $\vee$ are boolean false and disjunction respectively. Always is semantically equivalent to $\mathcal{G}$ in LTL.

Some derived forms are defined to capture the intention of typing.

$$[\psi] \stackrel{\text{def}}{=} \neg(!\ \psi), \qquad \text{After}^-\ \phi_1 : \phi_2 \stackrel{\text{def}}{=} \neg\text{After}^+\ \phi_1 : \neg\phi_2$$

$$\text{Before}^-\ \phi_1 : \phi_2 \stackrel{\text{def}}{=} \neg\text{Before}^+\ \phi_1 : \neg\phi_2$$

If we write down the formal meanings of $[\psi]$, $\text{After}^-\ \phi_1 : \phi_2$, and $\text{Before}^-\ \phi_1 : \phi_2$ following Definition 7, for any trace $w \in \Sigma^\infty$, we have

- $w \models [\psi]$ if and only if either (a) $w = \epsilon$, or (b) there exists $a \in \Sigma$ such that $a \preceq w$ and $a \Vdash \psi$.
- $w \models \text{After}^-\ \phi_1 : \phi_2$ if and only if there exists $u \preceq w$ such that $u \models \phi_1$, for any $u' \prec u$, $u' \not\models \phi_1$, and $w - u \models \phi_2$.
- $w \models \text{Before}^-\ \phi_1 : \phi_2$ if and only if either (a) there exists $u \preceq w$ such that $u \models \phi_1$, for any $u' \prec u$, $u' \not\models \phi_1$, and $u \models \phi_2$, or (b) for any $u \preceq w$, $u \not\models \phi_1$ and $w \models \phi_2$.

Thus, $[\psi]$ is a safety property violated only by a trace for which the first action does not satisfy the action formula ψ. $\text{Before}^-\ \phi_1 : \phi_2$ is intended to be a guarantee property, imposing both guarantee properties ϕ_1 and ϕ_2, and requires ϕ_2 to be discharged no later than ϕ_1. $\mathsf{After}^- \ \phi_1 : \phi_2$ is also intended to be a guarantee property, imposing a guarantee property ϕ_2 after the first instant when the guarantee property ϕ_1 is discharged.

Lastly, other derived forms are defined as follows.

$$\langle \mathtt{1} \rangle \overset{\text{def}}{=} \mathtt{true}, \qquad \langle \mathtt{k} \rangle \overset{\text{def}}{=} \mathsf{After}^- \ \langle \mathtt{1} \rangle : \langle \mathtt{k}-\mathtt{1} \rangle \quad (k \in \mathbb{N} \wedge k > 1)$$

$$\mathsf{Whenever} \ \phi_1 : \phi_2 \overset{\text{def}}{=} \mathsf{Always} \ \mathsf{After}^+ \ \phi_1 : \phi_2$$

$$\mathsf{Fulfilling}^k \ \phi_1 \,?\, \phi_2 : \phi_3 \overset{\text{def}}{=} (\mathsf{After}^+ \ \mathsf{Before}^- \ \langle \mathtt{k} \rangle : \phi_1 : \phi_2) \ \wedge$$
$$((\mathsf{Before}^+ \ \phi_1 : \neg\langle \mathtt{k}+\mathtt{1} \rangle) \vee \mathsf{After}^+ \ \langle \mathtt{k} \rangle : \phi_3) \quad (k \in \mathbb{N} \wedge k > 0)$$

$\langle \mathtt{1} \rangle$ is a guarantee property fulfilled by a trace of length one or more; $\langle \mathtt{k} \rangle$ extends the lower bound to k. $\mathsf{Whenever} \ \phi_1 : \phi_2$ imposes ϕ_2 whenever ϕ_1 holds. $\mathsf{Fulfilling}^k \ \phi_1 \,?\, \phi_2 : \phi_3$ asserts the following: if a guarantee property ϕ_1 is discharged within k steps (including the k^{th} step), then impose ϕ_2 to the suffix, otherwise impose ϕ_3.

In fact, in our policy language, brackets can be omitted without introducing any ambiguity for some formulae, for example, $\mathsf{After}^+ \ \phi_1 : (\mathsf{Eventually} \ \phi_2)$ can be written as $\mathsf{After}^+ \ \phi_1 : \mathsf{Eventually} \ \phi_2$ without ambiguity. Others, however, are required to resolve ambiguities. In order to omit unnecessary brackets, we define some binding priorities as follows:

Convention 8. *The unary connectives (*$\neg$*, *Eventually*, *Always*) bind most tightly. Next in the order comes binary boolean operators (*$\wedge$*, *$\vee$*), then comes binary temporal operators (*Before*, *After*, *Whenever*, *Ignoring*).*

Therefore, the formula $\mathsf{Before}^+ \ \phi_1 : (\phi_2 \wedge \phi_3)$ can be simplified as $\mathsf{Before}^+ \ \phi_1 : \phi_2 \wedge \phi_3$. However, brackets in the formula $(\mathsf{Before}^+ \ \phi_1 : \phi_2) \wedge \phi_3$ cannot be omitted. Sometimes, we might still write some unnecessary brackets to make the logic clear and explicit.

3.4 Semantic Equivalence

We also identify some interesting equivalences in our policy language as follows.

Proposition 9. *Given any* $w \in \Sigma^\infty$*, we have*

- $[\mathtt{true}] \equiv \top$
- $\mathtt{false} \equiv \bot$
- $w \models \mathtt{true}$ *if and only if* $w \neq \epsilon$
- $w \models [\mathtt{false}]$ *if and only if* $w = \epsilon$

Proposition 9 is rather obvious, so the proof is omitted.

Proposition 10.

$$\mathsf{Before}^+ \ \phi_1 : \phi_2 \equiv \mathsf{Before}^- \ \phi_1 : \phi_2 \tag{2}$$

Proof. To prove (2), we first prove $\neg\mathsf{Before}^+ \ \phi_1 : \phi_2 \equiv \mathsf{Before}^+ \ \phi_1 : \neg\phi_2$, which in fact, is to prove that for any trace w, $w \models \neg\mathsf{Before}^+ \ \phi_1 : \phi_2$ if and only if $w \models \mathsf{Before}^+ \ \phi_1 : \neg\phi_2$.

For "$\Rightarrow$". If $w \models \neg\mathsf{Before}^+ \ \phi_1 : \phi_2$, we have $w \not\models \mathsf{Before}^+ \ \phi_1 : \phi_2$, which implies one of the two cases: (a) there exists $u \preceq w$ such that $u \models \phi_1$, for all $u' \prec u$, $u' \not\models \phi_1$ but $u \not\models \phi_2$, namely,

$u \models \neg\phi_2$, hence implies $\mathsf{Before}^+\ \phi_1 : \neg\phi_2$, or (b) for all $u \preceq w$, $u \not\models \phi_1$, but $w \not\models \phi_2$, which also implies $\mathsf{Before}^+\ \phi_1 : \neg\phi_2$.

For "$\Leftarrow$". Suppose $w \models \mathsf{Before}^+\ \phi_1 : \neg\phi_2$. According to the semantics, this means that either (a) there exists $u \preceq w$, such that $u \models \phi_1$, for all $u' \prec u$, $u' \not\models \phi_1$, and $u \models \neg\phi_2$, namely, $u \not\models \phi_2$, hence implies $\neg\mathsf{Before}^+\ \phi_1 : \phi_2$, or (b) for all $u \preceq w$, $u \not\models \phi_1$ and $w \models \neg\phi_2$, or in another word $w \not\models \phi_2$, which also implies $\neg\mathsf{Before}^+\ \phi_1 : \phi_2$.

On the whole, we proved that $w \models \neg\mathsf{Before}^+\ \phi_1 : \phi_2$ if and only if $w \models \mathsf{Before}^+\ \phi_1 : \neg\phi_2$. Since $\mathsf{Before}^-\ \phi_1 : \phi_2$ is defined by $\neg\mathsf{Before}^+\ \phi_1 : \neg\phi_2$, and $\neg\mathsf{Before}^+\ \phi_1 : \neg\phi_2$ is equivalent to $\mathsf{Before}^+\ \phi_1 : \neg\neg\phi_2$, which, obviously is $\mathsf{Before}^+\ \phi_1 : \phi_2$. □

Note that although Before^+ and Before^- are semantically equivalent, we keep both operators in the language to capture the intention of typing (i.e., the superscript "+" is intended for safety properties and "−" is intended for guarantee property).

Proposition 11.

$$\neg\mathsf{Ignoring}\ \psi : \phi \equiv \mathsf{Ignoring}\ \psi : \neg\phi \tag{3}$$

Proof. The proof to this proposition is straightforward. For any trace w, $w \models \neg\mathsf{Ignoring}\ \psi : \phi$ if and only if $w \not\models \mathsf{Ignoring}\ \psi : \phi$, according to Clause 15 of Definition 7, if and only if $w|_\psi \not\models \phi$, which can be also written as $w|_\psi \models \neg\phi$. Therefore, $w \models \neg\mathsf{Ignoring}\ \psi : \phi$ if and only if $w \models \mathsf{Ignoring}\ \psi : \neg\phi$. □

3.5 Case Studies

After defining the syntax and semantics for the language, the next question would be: how can we adopt this language to express typical obligation policies?

In this section, we illustrate the use of our language through some case studies. First, an example of library loan policy is presented to show that the standard components of obligation policies can be captured by our language in a modular and composable manner. Then, some examples of obligation policies in application domain are formalized in our language to demonstrate the usefulness of our language.

3.5.1 Library Loan Policy

A common scenario of a library loan policy can be described as follows. If someone borrows a book from the library, she has to return the book before some deadline, otherwise, she will be penalized for the overdue book.

Policy 12. *Whenever a book is checked out from the library, one has to return the book within 30 days. Failing to fulfill this obligation, she will be charged \$1/day for the overdue book.*

Although such an obligation policy is not software-based, as here we only use it to demonstrate that our policy language provides means to mix and match the components summarized in Section 3.1, let us assume action propositions could also capture human behavior.

Suppose p_{cout}, p_{ret} and p_{fine} are action propositions satisfied by actions that check out a book, return a book and charge the fine respectively, and time step is count by day. Then,

- the ***trigger*** in this policy is the action of checking out a book, written as $\mathsf{Eventually}\ p_{cout}$.
- the ***obligation*** in this policy is the action of returning a book $\mathsf{Eventually}\ p_{ret}$, which is bounded by the ***temporal constraint*** with a deadline of 30 days $\langle 30 \rangle$.

- If the obligation is not fulfilled within 30 days, a fine will be charged before the book is returned. Then, the ***penalty*** would be $\mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{ret}\ :\ \mathsf{Always}\ p_{fine}$. There is no ***reward*** for fulfilling this obligation, thus it is merely a boolean constant $\top$.

As a result, combining each component with proper keywords, we are able to formalize the policy as follows.

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{cout}\ :$$

$$\mathsf{Fulfilling}^{30}\,(\mathsf{Before}^{-}\ \langle 30\rangle\ :\ \mathsf{Eventually}\ p_{ret})\,?$$

$$\top\ :\ (\mathsf{Before}^{+}\ \mathsf{Eventually}\,p_{ret}\ :\ \mathsf{Always}\ p_{fine})$$

In fact, there is a generic template in our language for expressing common idioms of obligation policies. Suppose ϕ_{trig}, ϕ_{obl}, ϕ_s, ϕ_e, ϕ_{reward} and $\phi_{penalty}$ correspond to the trigger, the obligation, the starting instant, the ending instant, the reward and the penalty respectively. Then a typical obligation policy is of the form

$$\mathsf{Whenever}\ \phi_{trig}\ :\ \mathsf{Fulfilling}^{k}\,(\mathsf{After}^{-}\ \phi_s\ :\ \mathsf{Before}^{-}\ \phi_e\ :\ \phi_{obl})\,?\,\phi_{reward}\ :\ \phi_{penalty}$$

Such a template states: whenever ϕ_{trig} holds, ϕ_{obl} shall be imposed within the time window $[t_s, t_e]$, where t_s and t_e are time instants identified by ϕ_s and ϕ_e. If such an obligation is fulfilled, ϕ_{reward} shall be imposed as reward, otherwise $\phi_{penalty}$ shall be imposed.

3.5.2 Software-based Obligation Policies

Next, let's look at some examples of obligation policies that are defined in software domain.

Policy 13. (File check-out and return [15]) *In a code versioning tool, a developer who checks out a file is obliged to check the file back in.*

Suppose the action propositions p_{cout} and p_{cin} are satisfied by file-check-out and file-check-in operations respectively.

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{cout}\ :$$

$$(\mathsf{Eventually}\ p_{cin} \wedge \mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{cin}\ :\ \mathsf{Always}\ \neg p_{cout})$$

The formula above demands that, once the file is checked out, no other check-out is allowed before the file is checked-in again.

Policy 14. (Logging following access denial [15]) *When a user attempts to access a document for which he lacks the required credentials, every subsequent attempts to access documents by that user must be logged.*

Suppose the action propositions p_{unauth_acc}, p_{acc} and p_{log} are satisfied respectively by actions that access documents without credentials, access to documents (with or without credentials) and log the subsequent document-access attempt of that user.

$$\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{unauth\_acc}\ :$$

$$((\mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{log}\ :\ \mathsf{Always}\ \neg p_{acc})\ \wedge$$

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{acc}\ :\ \mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{log}\ :\ \mathsf{Always}\ \neg p_{acc})$$

The formula above states that: after the first unauthorized access to a document, any subsequent document-access operations are disallowed before a log operation has been performed, and meanwhile, there is no two consecutive document-access operations without a log operation between them.

Policy 15. (Secured connecting after reading [1]) *An application can only open a file for reading. Once the file has been accessed, the application has to obtain approval from the user each time a connection is to be opened.*

Suppose the action propositions p_{write}, p_{read}, p_{acq_appr} and p_{conn} are satisfied by program actions that access a file for write, access a file for read, acquire user approval to connect to Internet, and make Internet connection respectively. Then the above policy has two requirements, which are:

1. writing to a file is not allowed, which can be specified as $\mathsf{Always}\ \neg p_{write}$, and
2. after a file has been read, the application is not allowed to connect to Internet before it has acquired approval from the user,

$$\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{read}\ :$$

$$((\mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{acq\_appr} : \mathsf{Always}\ \neg p_{conn})\ \wedge$$

$$(\mathsf{Whenever}\ \mathsf{Eventually}\ p_{conn} : \mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{acq\_appr} : \mathsf{Always}\ \neg p_{conn}))$$

Note that accessing a file for read is the ***trigger*** in this policy, but such a trigger can only be fired once.

Policy 16. (Secured PIM access [1]) *An application must not access the Personal Information Manager (PIM) while unsecured connections are open, and it must only open secure connections after the PIM is accessed.*

Suppose the action propositions p_{unsec_conn}, p_{close_conn} and p_{acc_pim} are satisfied by program actions that make a HTTP connection [3], close the opened HTTP connection, and access the PIM respectively. Then, similar to the previous one, this policy also has two requirements, which are:

1. whenever an unsecured Internet connection is established, before it is closed, the application is not allowed to access the PIM;

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{unsec\_conn}\ :$$

$$\mathsf{Before}^{+}\ \mathsf{Eventually}\ p_{close\_conn} : \mathsf{Always}\ \neg p_{acc\_pim}$$

2. after the application has accessed the PIM, unsecured Internet connection is always disallowed.

$$\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{acc\_pim} : \mathsf{Always}\ \neg p_{unsec\_conn}$$

Policy 17. (Cautious access to critical documents) *An application cannot request for user approval to access a critical document before its previous access to documents has been logged immediately. Once the application accesses a document without logging its access, it is no longer allowed to access critical file.*

[3]In contrast to the HTTPS connection, the HTTP connection are considered insecure.

Suppose the action propositionsp_{appr}, $p_{critical\_acc}$, p_{acc} and p_{log} are satisfied by program actions for acquiring user approval to access critical documents, accessing critical documents, accessing (critical or non-critical) documents, logging the document-access operations respectively. Then

1. without user approval, the application is not allowed to access critical documents.

$$\textsf{Before}^{+}\ \textsf{Eventually}\ p_{appr} : \textsf{Always}\ \neg p_{critical\_acc}$$

2. there is no two consecutive critical-document-access operations without a user-approval-acquiring operation in between.

$$\textsf{Whenever Eventually}\ p_{critical\_acc}\ :$$

$$\textsf{Before}^{+}\ \textsf{Eventually}\ p_{appr} : \textsf{Always}\ \neg p_{critical\_acc}$$

3. before the application demonstrates that it fulfills the logging-following-access (which implies that it has to perform document-access operation at least once), it is not allowed to acquire user approval to access the critical documents.

$$\textsf{Before}^{+}\ \textsf{Eventually}\ p_{acc} : \textsf{Always}\ \neg p_{appr}$$

4. if the application fulfills the obligation that it immediately logs its generic document-access operation, as a ***reward***, it unlocks its privilege to acquire user approval, but such a privilege is granted in a constrained way (i.e. when it acquires the user approval to access critical documents, it has to perform the critical-document-access operation). If such an obligation is violated, the application is not allowed to acquire user approval to access critical documents.

$$\textsf{Whenever Eventually}\ p_{acc}\ :$$

$$\textsf{Fulfilling}^{1} p_{log}\ ?$$

$$(\textsf{After}^{+}\ \textsf{Eventually}\ p_{appr}\ :$$

$$\textsf{Before}^{+}\ \textsf{Eventually}\ p_{critical\_acc} : \textsf{Always}\ \neg p_{appr})\ :$$

$$(\textsf{Always}\ \neg p_{appr})$$

Therefore, an application enforcing this policy can only behave in the following way: initially it is only allowed to access non-critical documents. If it performs logging immediately after its access, it is allowed to acquire user approval to access critical documents, otherwise the application will lose its chance forever to acquire user approval to access critical files. Furthermore, every critical-document-access operation has to be user approved, and every user approval acquired by the application has to be consumed before requesting for a new one.

The intention of this policy is as follows. Critical documents play a role as critical resources, which is not open to every application until the application demonstrates that it fulfills certain obligation. Once the application violates the obligation, it will be penalized with deprival of rights to access critical resources. Hence, it continuously protects the critical resources from being abused.

4 Recognize Enforceable and Monitorable Obligation Policies

In Section 3, we have seen how obligation policies can be specified in our policy language, which has a firm semantic foundation. We will now define a type system, in which the enforceability of these obligation policies can be checked.

Among the properties of obligation policies, there are two families of properties that are particularly interesting to us, which are

- non-empty safety property (i.e., ϵ always satisfies the property): the violation of the property can be detected by a reference monitor.
- non-universal guarantee property (i.e., ϵ never satisfies the property): the fulfillment of the property can be detected by a reference monitor.

In the type system, we adopt the type label ***enf*** to type a non-empty safety property. The implication is that the property can be enforced by a reference monitor. Such a property is also said to be ***enforceable***. Similarly, we adopt the type label ***mon*** to type a non-universal guarantee property. The implication is that the discharging of the property can be detected by a reference monitor. Such a property is also said to be ***monitorable***.

This section is organized as follows. Section 4.1 defines the inference rules of the type system. They are used to syntactically identify whether an obligation policy (written in our policy language) is enforceable or monitorable. Section 4.2 employs a case study to show how the type system can be used to guide a policy developer in formulating an enforceable obligation policy.

4.1 Type System

The type system is defined with the set of inference rules shown in Figure 4. Each inference rule defines the typing assertion $\phi : T$, where T is the type of ϕ. If T is of the form k-***enf***, where $k \in \mathbb{N} \cup \{\omega\}$, then ϕ is recognized to be a non-empty safety property. If $k \in \mathbb{N}$ then ϕ is a k-bounded safety property, meaning that the reference monitor can stop monitoring if no violation is detected in the first k steps. If T is of the form k-***mon***, where $k \in \mathbb{N} \cup \{\omega\}$, then ϕ is recognized to be a non-universal guarantee property. If $k \in \mathbb{N}$ then ϕ is a k-bounded guarantee property, meaning that the reference monitor can safely stop monitoring if the property is not fulfilled in the first k steps. The soundness of the type system is ensured by Theorem 30.

Note that the type system does not give us completeness, which means that there might exist enforceable or monitorable formulae in our policy language, but not detectable by the type system. However, some formulae are intended to be ruled out by our type system for one of the following two reasons.

Formula that is nonsensical. Suppose ϕ is a generic formula, ϕ_{enf} and ϕ_{mon} are enforceable and monitorable formulae respectively. Then, some compositions that betray the intention of typing will generate nonsensical formulae. Examples are $\mathsf{After}^{+}\ \phi_{enf} : \phi$, $\mathsf{After}^{-}\ \phi_{enf} : \phi$, $\mathsf{Before}^{+}\ \phi_{enf} : \phi$, $\mathsf{Before}^{-}\ \phi_{enf} : \phi$, $\mathsf{Before}^{+}\ \phi_{mon} : \phi_{mon}$ and $\mathsf{Before}^{-}\ \phi_{mon} : \phi_{enf}$.

Consider $\mathsf{After}^{+}\ \phi_{enf} : \phi$ for example. A trace w satisfies $\mathsf{After}^{+}\ \phi_{enf} : \phi$ if and only if either (a) every prefix of w violates ϕ_{enf}, or (b) there exists a shortest prefix satisfies ϕ_{enf}, and ϕ holds afterward. Assume the soundness of typing, ϕ_{enf} is a non-empty safety property, which means, it is always satisfied by ϵ, so (a) does not hold. Similarly, the shortest prefix satisfying ϕ_{enf} would be ϵ, thus $\mathsf{After}^{+}\ \phi_{enf} : \phi$ imposes only ϕ on w. On the whole, $\mathsf{After}^{+}\ \phi_{enf} : \phi$ merely is ϕ.

As proved in Section 3.4, Before^{+} and Before^{-} are semantically equivalent, and we keep both operators in the language to highlight the intended usage of the two constructs (i.e., + for safety,

$$\psi : 1\text{-}\boldsymbol{mon} \quad (\text{TM-Ax}) \qquad [\psi] : 1\text{-}\boldsymbol{enf} \quad (\text{TE-Ax})$$

$$\frac{\phi : k\text{-}\boldsymbol{enf}}{\neg\phi : k\text{-}\boldsymbol{mon}} \quad (\text{TM-Ne}) \qquad \frac{\phi : k\text{-}\boldsymbol{mon}}{\neg\phi : k\text{-}\boldsymbol{enf}} \quad (\text{TE-Ne})$$

$$\frac{\phi : k\text{-}\boldsymbol{mon}}{\textsf{Eventually}\ \phi : \omega\text{-}\boldsymbol{mon}} \quad (\text{TM-Ev}) \qquad \frac{\phi : k\text{-}\boldsymbol{enf}}{\textsf{Always}\ \phi : \omega\text{-}\boldsymbol{enf}} \quad (\text{TE-Al})$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \\ k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \vee \phi_2 : k\text{-}\boldsymbol{mon}} \quad (\text{TM-Or}) \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{enf} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \\ k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \vee \phi_2 : k\text{-}\boldsymbol{enf}} \quad (\text{TE-Or})$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \\ k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \wedge \phi_2 : k\text{-}\boldsymbol{mon}} \quad (\text{TM-An}) \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{enf} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \\ k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \wedge \phi_2 : k\text{-}\boldsymbol{enf}} \quad (\text{TE-An})$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon}}{\textsf{Before}^{-}\ \phi_1 : \phi_2 : k_2\text{-}\boldsymbol{mon}} \quad (\text{TM-Be}) \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf}}{\textsf{Before}^{+}\ \phi_1 : \phi_2 : k_2\text{-}\boldsymbol{enf}} \quad (\text{TE-Be})$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \\ k = k_1 + k_2}{\textsf{After}^{-}\ \phi_1 : \phi_2 : k\text{-}\boldsymbol{mon}} \quad (\text{TM-Af}) \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \\ k = k_1 + k_2}{\textsf{After}^{+}\ \phi_1 : \phi_2 : k\text{-}\boldsymbol{enf}} \quad (\text{TE-Af})$$

$$\frac{\phi : k\text{-}\boldsymbol{mon}}{\textsf{Ignoring}\ \psi : \phi : \omega\text{-}\boldsymbol{mon}} \quad (\text{TM-Ig}) \qquad \frac{\phi : k\text{-}\boldsymbol{enf}}{\textsf{Ignoring}\ \psi : \phi : \omega\text{-}\boldsymbol{enf}} \quad (\text{TE-Ig})$$

$$\frac{k \in \mathbf{N}}{\langle \texttt{k} \rangle : k\text{-}\boldsymbol{mon}} \quad (\text{TM-Co}) \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad k_1 \in \mathbf{N} \\ \phi_2 : k_2\text{-}\boldsymbol{enf} \qquad \phi_3 : k_3\text{-}\boldsymbol{enf} \\ k = k_1 + \boldsymbol{max}\,(k_2,\ k_3)}{\textsf{Fulfilling}^{k_1}\ \phi_1\ ?\ \phi_2 : \phi_3 : k\text{-}\boldsymbol{enf}} \quad (\text{TE-Fu})$$

Figure 4: Inference rules for the type system

– for guarantee). Thus, it is meaningless to deviates from the intention, and having $\textsf{Before}^{+}\ \phi_{mon} : \phi_{mon}$ and $\textsf{Before}^{-}\ \phi_{mon} : \phi_{enf}$.

Formula that is neither enforceable nor monitorable property. Also, suppose ϕ is a generic formula, ϕ_{enf} and ϕ_{mon} are enforceable and monitorable formula respectively. There are some compositions, although meaningful, but generate formulae that are neither safety nor guarantee properties. Examples are $\phi_{enf} \wedge \phi_{mon}$, $\phi_{enf} \vee \phi_{mon}$, $\textsf{After}^{+}\ \phi_{mon} : \phi_{mon}$ and $\textsf{After}^{-}\ \phi_{mon} : \phi_{enf}$.

Consider $\textsf{After}^{-}\ \phi_{mon} : \phi_{enf}$ for example. A trace w satisfies $\textsf{After}^{-}\ \phi_{mon} : \phi_{enf}$ if and only if there exists a shortest prefix that satisfies ϕ_{mon}, and ϕ_{enf} holds afterward. Assume the soundness of typing. Obviously, ϵ does not satisfy the formula, hence it is not an enforceable property. But given a trace which has a prefix satisfying ϕ_{mon}, and not every extension to that prefix satisfies ϕ_{enf}, so the formula is not monitorable as well.

4.2 Case Study

As mentioned above, the type system is able to provide feedbacks to policy developers in formalizing enforceable obligation policies. In this section, we demonstrate this use of type system through a case study.

Consider the library loan policy again, but from the policy developers' prospective. Suppose a policy developer is to write a library loan policy, which states that every book checked out from the library has to be returned. The goal is to formulate the policy in such a way that it is enforceable. Let the action propositions p_{cout} and p_{ret} refer to check-out and return operations respectively. The time step is counted by day.

Let us start with a naive formulation that captures the main intention.

Policy 18. *Every book checked out from the library has to be returned.*

$$\mathsf{Whenever\ Eventually}\, p_{cout} \ : \ \mathsf{Eventually}\ p_{ret}$$

Note that Whenever is merely a shorthand for Always After$^+$. The derivation tree for typing this formula is shown in Figure 5. By TE-Af, the formula is not well-typed, meaning that the formula is not enforceable. TE-Af also identifies the sub-formula (Eventually p_{ret}) to be the cause (i.e., the sub-formula is monitorable rather than enforceable). The policy developer refines the policy as follows.

Policy 19. *Every book checked out from the library has to be returned, otherwise the book user will be penalized for not allowing to borrow any books from the library.*

$$\mathsf{Whenever\ Eventually}\ p_{cout} \ : \ \mathsf{Fulfilling}^k\ \mathsf{Eventually}\, p_{ret}\ ?\ \top \ : \ \mathsf{Always}\, \neg p_{cout}$$

The policy developer hopes that by protecting the Eventually clause with a Fulfilling clause, the latter can be typed enforceable by virtual of TE-Fu. Yet, the the formula is still not well typed because TE-Fu expects (Eventually p_{ret}) to be a k-bounded guarantee property for some finite k. The derivation tree for typing this formula is given in Figure 6. Thus, this policy clause must be refined further.

Policy 20. *Every book checked out from the library has to be returned within 30 days, otherwise the book user will be penalized for not allowing to borrow any book from the library.*

$$\begin{aligned}&\mathsf{Whenever\ Eventually}\ p_{cout}\ :\\ &\qquad \mathsf{Fulfilling}^{30}\, (\mathsf{Before}^-\ \langle 30\rangle : \mathsf{Eventually}\ p_{ret})\ ?\ \top \ : \ \mathsf{Always}\ \neg p_{cout}\end{aligned}$$

The policy developer now sets a deadline for the Eventually clause, making the obligation a bounded guarantee property. And the derivation tree for typing this formula is given in Figure 7. Apparently, the type system successfully recognizes this formula to have type ω-***enf***, meaning that the formula is enforceable.

Lastly, note that a reasonable implementation of the type checker checker can easily *infer* the both the bound in the Fulfilling construct and the signs (+ or −) of the Before and After constructs. So the policy developer can in theory omit these details.

$$\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{cout} : \omega\text{-}\boldsymbol{mon}}\,(\text{TM-Ev}) \qquad \dfrac{aProp_{ret} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{ret} : \omega\text{-}\boldsymbol{mon}}\,(\text{TM-Ev})}{\textsf{After}^{+}\ \textsf{Eventually } p_{cout} : \textsf{Eventually } p_{ret}}\,(\text{Err:TE-Af})}{\textsf{Whenever Eventually } p_{cout} : \textsf{Eventually } p_{ret}}\,(\text{TE-Al})$$

Figure 5: Derivation tree for typing Policy 18. (Erro:TE-Af) indicates that error occurs when applying rule TE-Af.

$$\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{cout} : \omega\text{-}\boldsymbol{mon}}\,(\text{TM-Ev}) \qquad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{ret} : \omega\text{-}\boldsymbol{mon}}\,(\text{TM-Ev}) \quad k_1 = \omega \quad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\neg p_{ret} : 1\text{-}\boldsymbol{enf}}\,(\text{TE-No})}{\textsf{Always } \neg p_{cout} : \omega\text{-}\boldsymbol{enf}}\,(\text{TE-Al})}{\textsf{Fulfilling}^{\omega}\ \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}}\,(\text{Err: TE-Fu})}{\textsf{After}^{+}\ \textsf{Eventually } p_{cout} : \textsf{Fulfilling}^{\omega}\ \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}}\,(\text{TE-Af})}{\textsf{Whenever Eventually } p_{cout} : \textsf{Fulfilling}^{\omega}\ \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}}\,(\text{TE-Al})$$

Figure 6: Derivation tree for typing Policy 19. (Err:TE-Fu) indicates that error occurs when applying rule TE-Fu.

$$\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{cout} : 1\text{-}\boldsymbol{mon}}\,(\text{TM-Ev}) \qquad \dfrac{\dfrac{\dfrac{30 \in \mathbb{N}}{\langle 30 \rangle : 30\text{-}\boldsymbol{mon}}\,(\text{TM-Co}) \quad \dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\textsf{Eventually } p_{ret} : \omega\text{-}\boldsymbol{mon}}\,(\text{TM-Ev})}{\textsf{Before}^{-}\ \langle 30 \rangle : \textsf{Eventually } p_{ret} : 30\text{-}\boldsymbol{mon}}\,(\text{TM-Be}) \quad k_1 = 30 \quad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\neg p_{ret} : 1\text{-}\boldsymbol{enf}}\,(\text{TE-No})}{\textsf{Always } \neg p_{cout} : \omega\text{-}\boldsymbol{enf}}\,(\text{TE-Al})}{(\textsf{Fulfilling}^{30}\ \textsf{Before}^{-}\ \langle 30 \rangle : \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}) : 30\text{-}\boldsymbol{enf}}\,(\text{TE-Fu})}{(\textsf{After}^{+}\ \textsf{Eventually } p_{cout} : \textsf{Fulfilling}^{30}\ \textsf{Before}^{-}\ \langle 30 \rangle : \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}) : \omega\text{-}\boldsymbol{enf}}\,(\text{TE-Af})}{(\textsf{Whenever Eventually } p_{cout} : \textsf{Fulfilling}^{30}\ \textsf{Before}^{-}\ \langle 30 \rangle : \textsf{Eventually } p_{ret} \mathrel{?} \top : \textsf{Always } \neg p_{cout}) : \omega\text{-}\boldsymbol{enf}}\,(\text{TE-Al})$$

Figure 7: Derivation tree for typing Policy 20

5 An Enforcement and Monitoring Mechanism

In this section, we define a representation of reference monitor, named ***Obligation Monitor (OM)***, as the enforcement and monitoring mechanism for formulae in our policy language. Partly inspired by [46], it is designed to facilitate the inlining of monitoring logic. The obligation monitor is fully capable of expressing deontic concepts such as rights, prohibitions, obligations and dispensations [29].

This section is organized as follows. In Section 6, we start with a formal definition of OM. Then, we discuss how the deontic concepts can be captured using the basic operations of OMs. Lastly, we describe why the OMs can be easily inlined in an untrusted program. Section 7 elaborates the expressiveness of OMs comparing to finitely-branching security automata.

6 Obligation Monitors

An OM can be used for representing either an enforcement mechanism for a safety property, or a monitoring mechanism for a guarantee property. More formally,

Definition 21. (Obligation Monitor) *An OM $\mathcal{M}$ is a quadruple of the form $\langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0\rangle$, where*

- $\mathcal{L}_\Sigma$ *is an* **action assertion language** *over the set of actions Σ.*
- $\mathcal{O}$ *is a countable set of* **obligation identifiers**.
- *The set $\mathsf{states}_\mathcal{M}$ of monitor states is defined as $[\mathcal{O}]^{<\omega}$. $S_0 \in [\mathcal{O}]^{<\omega}$ is an* **initial state**.
- $\iota : \mathcal{O} \to \mathsf{obl}_\mathcal{M}$ *assigns to each obligation identifier an* **obligation** *from $\mathsf{obl}_\mathcal{M}$, which is defined as follows:*

$$\mathsf{obl}_\mathcal{M} = \Phi(\mathcal{L}_\Sigma) \uplus \left(\Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}}\right)$$

  An obligation of the form $\psi \in \Phi(\mathcal{L}_\Sigma)$ is called a **simple condition**. *An obligation of the form $\langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del}\rangle \in \Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}}$ is called a* **trigger rule**, *where*

  - ψ *is the* **trigger condition**, *satisfaction of which causes the rule to be applicable to the monitor state,*
  - $\mathcal{O}_{add}$ *identifies a finite set of obligation identifiers to be added to the monitor state once this rule is applied, and,*
  - $\mathcal{O}_{del}$ *identifies a set of obligation identifiers to be deleted from the monitor state once this rule is applied.*

  Specifically, an OM that contains only trigger rules (i.e., no obligation identifier in the OM is mapped to a simple condition by ι) is also called a **Pure Obligation Monitor**.

Given a monitor state $S \in \mathsf{states}_\mathcal{M}$, let $\mathsf{cond}_\mathcal{M}(S) = \{o \in S \mid \iota(o) \in \Phi(\mathcal{L}_\Sigma)\}$ be the set of simple conditions contained in S. Similarly, $\mathsf{rule}_\mathcal{M}(S) = \{o \in S \mid \iota(o) \in \Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}}\}$ is defined as the set of trigger rules that appears in S. Let $\mathsf{trig}_\mathcal{M}(S, a) = \{o \in \mathsf{rule}_\mathcal{M}(S) \mid \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del}\rangle \wedge (a \Vdash \psi)\}$ be the set of trigger rules that are triggered by a at the monitor state S. Moreover, $\mathsf{add}_\mathcal{M}(S, a)$ and $\mathsf{del}_\mathcal{M}(S, a)$ are defined as the sets of obligation identifiers that will be added into and deleted from S after applying all the rules fired by a at the monitor state. More formally,

$$\mathsf{add}_{\mathcal{M}}(S,a) = \bigcup \{\, \mathcal{O}_{add} \mid o \in \mathsf{trig}_{\mathcal{M}}(S,a),\ \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \,\}$$
$$\mathsf{del}_{\mathcal{M}}(S,a) = \bigcup \{\, \mathcal{O}_{del} \mid o \in \mathsf{trig}_{\mathcal{M}}(S,a),\ \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \,\}$$

Therefore, we define a state transition relation $\cdot \xrightarrow{\cdot}_{\mathcal{M}} \cdot \subseteq \mathsf{states}_{\mathcal{M}} \times \Sigma \times \mathsf{states}_{\mathcal{M}}$ as follows. Given an action $a \in \Sigma$, a transition $S_1 \xrightarrow{a}_{\mathcal{M}} S_2$ can be made if and only if both of the following hold:

1. for every $o \in \mathsf{cond}_{\mathcal{M}}(S_1)$, we have $a \Vdash \iota(o)$,
2. $S_2 = (S_1 \cup \mathsf{add}_{\mathcal{M}}(S_1, a)) \setminus \mathsf{del}_{\mathcal{M}}(S_1, a)$.

We can extend the one-step transition relation to describe transitions generated by sequences of actions. It is defined inductively as follows: let $S \xrightarrow{\epsilon}_{\mathcal{M}} S$; for $a \in \Sigma$ and $w \in \Sigma^*$, we write $S_1 \xrightarrow{a \cdot w}_{\mathcal{M}} S_2$ whenever $S_1 \xrightarrow{a}_{\mathcal{M}} S_3$ and $S_3 \xrightarrow{w}_{\mathcal{M}} S_2$ for some state $S_3 \in \mathsf{states}_{\mathcal{M}}$.

We say that the OM $\mathcal{M}$ ***accepts*** the sequence $w \in \Sigma^\infty$ iff for every $w' \preceq w$, $S_0 \xrightarrow{w'}_{\mathcal{M}} S$ for some $S \in \mathsf{states}_{\mathcal{M}}$; and it ***recognizes*** the sequence $w \in \Sigma^\infty$ iff there exists $w' \preceq w$, $S_0 \xrightarrow{w'}_{\mathcal{M}} \emptyset$. We write $\mathcal{L}_{acc}(\mathcal{M})$ to denote the set of all action sequences accepted by $\mathcal{M}$. An OM $\mathcal{M}$ ***enforces*** a property $\mathcal{L}$ iff $\mathcal{L} = \mathcal{L}_{acc}(\mathcal{M})$. Such a property is called ***OM-enforceable***. Similarly, we write $\mathcal{L}_{rec}(\mathcal{M})$ to denote the set of all action sequences recognized by $\mathcal{M}$. An OM $\mathcal{M}$ ***monitors*** a property $\mathcal{L}$ iff $\mathcal{L} = \mathcal{L}_{rec}(\mathcal{M})$. Such a property is called ***OM-monitorable***. Note that $\mathcal{L}_{rec}(\mathcal{M}) \subseteq \mathcal{L}_{acc}(\mathcal{M})$, which means that trace recognition implies trace accepting.

The four policy objects (i.e. rights, prohibitions, obligations and dispensations) in Rei [29] can be captured using the simple conditions and trigger rules of OMs in the following way.

- Rights are permissions for an entity to perform certain action. These reflect to simple conditions in an OM: i.e., an action is permitted to be performed at a monitor state iff it satisfies all the simple conditions in that state.
- Prohibitions are negative authorizations, which implies that an entity cannot perform certain action. Thus, they are merely mirror images of rights, which could also be captured by simple conditions: i.e., an action cannot be performed at a monitor state iff it does not satisfy one of the simple conditions in that state.
- Obligations are actions that an entity must perform and are usually triggered when a certain set of conditions are true. They could be captured by trigger rules in an OM: i.e., the trigger rules can be chained in such a way that firing one rule will add another rule into the monitor state, which indicates that there are some actions are putting on the waiting list to be performed (since if an OM is clear of obligations, it will enter an empty state).
- Dispensations are actions that an entity is no longer required to perform. They are symmetrical to obligations, thus they could also be captured using trigger rules: i.e., firing a trigger rule at a monitor state could also cause another rule to be deleted from the state, which indicates that there are some actions that are no longer required to be performed.

The OM contains only a limited set of state-altering primitives — adding obligation identifiers to or deleting obligation identifiers from the monitor state. These primitive operations can be easily injected into the target program. Therefore, it is anticipated that such a representation of reference monitor facilitates the inlining of program monitors. Moreover, as its structure is similar to the policy representation proposed by Fei and Fong [46], it is conjectured that the optimization through constant propagation and liveness analysis can also be applicable to OMs.

7 Expressiveness of Obligation Monitor

To measure the expressiveness of obligation monitors, a variant of security automata is defined here. Given the set of actions Σ, a ***Deterministic Security Automaton (DSA)*** is a triple $\mathcal{N} = \langle Q, q_0, \delta_{a\in\Sigma}\rangle$

- Q is a countable set of states.
- $q_0 \in Q$ is an initial state.
- $\delta_{a\in\Sigma}$ is an indexed family of partial transition functions $\delta_a : Q \rightharpoonup Q$.

We extend the transition functions for sequences of actions. Specifically, for $a \in \Sigma$ and $w \in \Sigma^*$, we have $\delta_\epsilon = i_Q$ and $\delta_{a\cdot w} = \delta_w \circ \delta_a$, where i_Q is the total identity function on Q, and $\circ$ is function composition. An action sequence $w \in \Sigma^\infty$ is ***accepted*** by the DSA iff $\delta_w(q_0)$ is defined. We write $\mathcal{L}(\mathcal{N})$ to denote the set of all action sequences accepted by $\mathcal{N}$. A safety property $\mathcal{L}$ is ***enforced*** by a DSA $\mathcal{N}$ if and only if $\mathcal{L} = \mathcal{L}(\mathcal{N})$.

A ***Finitely-Branching DSA (FB-DSA)*** is a DSA for which every state has only a finite number of successor states. Note that it is still possible that an infinite number of actions are applicable in a given state. Similarly, a safety property $\mathcal{L}$ is ***FB-DSA-enforceable*** if and only if it is enforceable by some FB-DSA.

In the remainder of this section, we try to compare the FB-DSA with OMs in terms of enforcing and monitoring properties.

7.1 Enforcing Safety Properties

It is not hard to identify that $\mathcal{L}_{acc}(\mathcal{M})$ is prefix-closed and non-empty, so $\mathcal{L}_{acc}(\mathcal{M})$ is a non-empty safety property. OMs have the same expressive power as the FB-DSA in enforcing safety properties.

Theorem 22. *A safety property is OM-enforceable iff it is FB-DSA-enforceable.*

Proof. In this proof, we show that, given any OM $\mathcal{M}$, one can construct an equivalent FB-DSA $\mathcal{N}$ such that $\mathcal{L}_{acc}(\mathcal{M}) = \mathcal{L}(\mathcal{N})$, and vice versa.

The forward direction is relatively straightforward to demonstrate. An OM is obviously equivalent to a FB-DSA by treating the OM's states as DSA states and constructing an appropriate family of transition functions. Note that an OM state always has finitely many successors. To see this, since an OM state contains finitely many obligation identifiers, and suppose all identifiers are trigger rules. Each rule may or may not be triggered by an action. There is thus finitely many different ways in which the trigger rules may be triggered, and thus as many successor states.

We now demonstrate the backward direction. Consider a FB-DSA $\mathcal{N} = \langle Q, q_0, \delta_{a\in\Sigma}\rangle$. For each $q \in Q$, denote $\{q' \in Q \mid \exists a \in \Sigma.\, \delta_a(q) = q'\}$ by $\mathsf{succ}(q)$. Let $\mathcal{P}$ be the countable set $\{p_q \mid q \in Q\} \cup \{p_{q_1,q_2} \mid q_1 \in Q \wedge q_2 \in \mathsf{succ}(q_1)\}$, and define $\Vdash$ as follows:

$$a \Vdash p_q \text{ iff } \delta_a(q) \text{ is defined}$$
$$a \Vdash p_{q_1,q_2} \text{ iff } \delta_a(q_1) = q_2$$

Let $\mathcal{O}$ be the countable set $\{o_q \mid q \in Q\} \cup \{o_{q_1,q_2} \mid q_1 \in Q \wedge q_2 \in \mathsf{succ}(q_1)\}$. Define S_q to be the set $\{o_q\} \cup \{o_{q,q'} \mid q' \in \mathsf{succ}(q)\}$, and define ι as follows:

$$\iota(o_q) = p_q$$
$$\iota(o_{q_1,q_2}) = \langle p_{q_1,q_2}, S_{q_2}, S_{q_1}\rangle$$

Now, consider the OM $\mathcal{M}_\mathcal{N} = \langle \mathcal{P}, \mathcal{O}, \iota, S_{q_0}\rangle$. It is easy to see that $\mathcal{L}(\mathcal{N}) = \mathcal{L}_{acc}(\mathcal{M}_\mathcal{N})$. $\square$

7.2 Monitoring Guarantee Properties

Symmetrically, if an OM recognizes a sequence, it also recognizes every extension of that sequence. Consequently, $\mathcal{L}_{rec}(\mathcal{M})$ is a guarantee property. Due to the duality of guarantee property and safety property, given Theorem 22, the following corollary is true.

Corollary 23. *A guarantee property is OM-monitorable iff its complement is FB-DSA-enforceable.*

By Corollary 23, the OM is as expressive as FB-DSA in monitoring guarantee properties. In addition, the following propositions hold for the OM when monitoring a guarantee property.

Proposition 24. *Suppose $\mathcal{L}$ is a guarantee property. If $\mathcal{L}$ is monitored by some OM, then $\mathcal{L}$ is also monitored by some pure OM.*

This proposition legitimizes a certain style of compilation, in which monitorable formulae are compiled into pure OMs. On one hand, this enables us to link between the structures of OMs and the classes of properties the OMs are to cope with (i.e., pure OM for monitoring monitorable properties and non-pure OM for enforcing enforceable properties). On the other hand, this is necessary for the compilation of some formulae, for example $\mathsf{Before}^+\ \phi_1 : \phi_2$ (a detailed compilation will be presented in Section 8.1). It requires the cooperation and communication of several sub-OMs. With all monitorable formulae compiled into pure OMs, it enables us to claim that a trace is rejected by an OM iff it violates an enforceable property. Proposition 24 is proved as follows.

Proof. Given the same set Σ of actions, suppose $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S_0' \rangle$ is an OM that monitors $\mathcal{L}$. We construct the required pure OM as follows. Let o^* be an obligation identifier such that $o^* \notin \mathcal{O}'$. Then, we define ι as follows.

$$\iota(o) = \begin{cases} \langle \mathtt{true}, \emptyset, \emptyset \rangle & o = o^* \\ \iota'(o) & \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle !\ \psi, \{o^*\}, \emptyset \rangle & \iota'(o) = \psi \end{cases}$$

Intuitively, ι and ι' are identical for trigger rules in $\mathcal{M}'$. For simple conditions in $\mathcal{M}'$, ι re-interprets them as trigger rules, which are fired by actions violating the simple conditions. Once fired, these trigger rules will add o^* into the monitor state. The obligation identifier o^* is mapped by ι to a trigger rule that will be fired by any action, but will not add or delete obligation identifiers from the monitor state. As $o^* \notin \mathcal{O}'$, once o^* is added into the monitor state, it cannot be deleted by any trigger rules from $\mathcal{M}'$. Therefore, for the pure OM $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \{o^*\}, \iota, S_0' \rangle$, it is obvious that $\mathcal{L}_{rec}(\mathcal{M}) = \mathcal{L}_{rec}(\mathcal{M}')$. $\square$

Proposition 25. *Suppose $\mathcal{L}$ is a non-universal guarantee property. If $\mathcal{L}$ is monitored by some pure OM, then $\mathcal{L}$ is also monitored by some pure OM $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$ for which there exists a set $\mathcal{O}_f \subseteq \mathcal{O}$ such that:*

$$\forall w \in \Sigma^*. \forall S \in \mathsf{states}_{\mathcal{M}}. \left(S_0 \xrightarrow{w}_{\mathcal{M}} S_1 \ \wedge\ S_1 \neq \emptyset \right) \Rightarrow$$
$$\forall a \in \Sigma. \forall S \in \mathsf{states}_{\mathcal{M}}. \left(\left(S_1 \xrightarrow{a}_{\mathcal{M}} S_2 \ \wedge\ S_2 = \emptyset \right) \Leftrightarrow \mathsf{trig}_{\mathcal{M}}(S_1, a) \cap \mathcal{O}_f \neq \emptyset \right)$$

The trigger rules identified in $\mathcal{O}_f$ are called ***final trigger rules***. Note that a trace is recognized by a pure OM iff it puts the pure OM to an empty state, so sequence recognition coincides with the firing of final trigger rules. In other words, Proposition 25 in fact specifies that given a pure OM which monitors a guarantee property $\mathcal{L}$, the firing of final trigger rules implies the fulfillment of $\mathcal{L}$.

Proof. Suppose $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$ is a pure OM that monitors $\mathcal{L}$. I construct the required pure OM $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S_0' \rangle$ as follows.

Consider state $S \in [\mathcal{O}]^{<\omega}$. There are finitely many trigger rules in S. A trigger rule in S is either triggered or not. So there are at most $2^{|S|}$ next states for state S. In addition, $\mathsf{states}_\mathcal{M}$ is countable.

Define $\mathcal{O}'$ to be the countable set $\{o_{S_1,S_2} \mid S_1, S_2 \in \mathsf{states}_\mathcal{M} \wedge (\exists w \in \Sigma^* . S_0 \xrightarrow{w}_\mathcal{M} S_1) \wedge S_1 \neq \emptyset \wedge (\exists a \in \Sigma . S_1 \xrightarrow{a}_\mathcal{M} S_2)\}$, and define ι' as follows:

$$\iota'(o_{S_1,S_2}) = \langle \psi_{S_1,S_2}, \mathcal{O}_{S_2}, \mathcal{O}_{S_1} \rangle$$

where

$$\mathcal{O}_{S_1} = \begin{cases} \emptyset & \text{if } S_1 = \emptyset \\ \{o_{S_1,S_2} \mid \exists a \in \Sigma . S_1 \xrightarrow{a}_\mathcal{M} S_2\} & \text{otherwise} \end{cases}$$

We now construct ψ_{S_1,S_2} using the following procedure.

Suppose $\mathcal{O}$ is a set of trigger rules. Define $\mathtt{add}(\mathcal{O})$ and $\mathtt{del}(\mathcal{O})$ as follows:

$$\mathtt{add}(\mathcal{O}) = \bigcup \{\mathcal{O}_{add} | o \in \mathcal{O} \wedge \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle\},$$
$$\mathtt{del}(\mathcal{O}) = \bigcup \{\mathcal{O}_{del} | o \in \mathcal{O} \wedge \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle\}$$

Let $\Pi_{S_1,S_2} = \{\mathcal{O}_i^* \mid (\mathcal{O}_i^* \subseteq S_1) \wedge (S_2 = (S_1 \cup \mathtt{add}(\mathcal{O}_i^*)) \backslash \mathtt{del}(\mathcal{O}_i^*))\}$. Suppose $\mathcal{O}_i^* = \{o_1, o_2, \cdots o_m\} \in \Pi_{S_1,S_2}$. Suppose further $S_1 \backslash \mathcal{O}_i^* = \{o_1', o_2', \cdots, o_n'\}$. Then define:

$$\psi_i^* = \psi_1 \,\&\&\, \psi_2 \,\&\& \cdots \&\&\, \psi_m \,\&\&\, !\, \psi_1' \,\&\&\, !\, \psi_2' \,\&\& \cdots \&\&\, !\, \psi_n'$$

Suppose $\Pi_{S_1,S_2} = \{\mathcal{O}_1^*, \mathcal{O}_2^*, \cdots, \mathcal{O}_N^*\}$. Note that for each $\mathcal{O}_i^* \in \Pi_{S_1,S_2}$, there is a corresponding ψ_i^*. Then,

$$\psi_{S_1,S_2} = \psi_1^* \,||\, \psi_2^* \,||\, \cdots \,||\, \psi_N^*$$

Now, define $S_0' = \mathcal{O}_{S_0'}$. We have for the OM $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S_0' \rangle$, $\mathcal{L}_{rec}(\mathcal{M}') = \mathcal{L}_{rec}(\mathcal{M})$. Lastly, if we take $\mathcal{O}_f' = \{o_{S_1,S_2} \in \mathcal{O} \mid S_2 = \emptyset\}$, then the trigger rules identified in $\mathcal{O}_f'$ are triggered iff the resulting state is $\emptyset$. □

Later in Section 8, when an OM requires the cooperation and communication of several sub-OMs, the final trigger rules of a sub-OM can be used to inform another sub-OM the fulfillment of a monitorable property.

8 Policy Compilation

In Section 5, we defined the obligation monitor, which represents either an enforcement mechanism for safety properties, or a monitoring mechanism for guarantee properties. In this section, I present a compilation algorithm, which compiles the well-typed formulae in our policy language to the OMs.

This section is organized as follows. In Section 8.1, I describe a compilation algorithm. Then section 8.2 reports some examples of compilations. Finally, a theorem is presented and proved in Section 8.3. This theorem ensures the correctness of the compilation algorithm, and also entails the soundness of the type system introduced in Section 4.

8.1 Compilation Algorithm

We start with a formal definition of the interface of the compilation algorithm as follows:

Definition 26. *The compilation algorithm* `compile`(ϕ)*, takes obligation policy* ϕ *as argument, and returns a tuple* $\langle \mathcal{T}, \mathcal{B}, \mathcal{M}, \mathcal{O}_f \rangle$*, where:*

- $\mathcal{T} \in \{\boldsymbol{enf}, \boldsymbol{mon}\}$ *is the type of* ϕ *assigned by the type system;*
- $\mathcal{B} \in \mathbb{N} \cup \{\omega\}$ *is the bound of* ϕ *assigned by the type system;*
- $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$ *is an OM, which enforces* ϕ *if* $\mathcal{T} = \boldsymbol{enf}$*, or monitors* ϕ *if* $\mathcal{T} = \boldsymbol{mon}$*;*
- $\mathcal{O}_f \subseteq \mathcal{O}$ *is the set of final trigger rules for* $\mathcal{M}$ *if* $\mathcal{T} = \boldsymbol{mon}$*.* $\mathcal{O}_f = \emptyset$ *if* $\mathcal{T} = \boldsymbol{enf}$*.*

Note that the formulae passed into the `compile` interface are assumed to be well-typed, which is ensured by the embedded type checker implemented in the compiler. The compilation is syntax directed, so in order to compile a composite formula, the compilations for its sub-formulae are used as building blocks. For example, to compile the formula $\neg\phi'$, the compilation for its sub-formula ϕ' will be used.

According to Section 3, there are a set of primitive operators, as well as derived forms defined in the policy language. Intuitively, the compilations for a formula with derived operators could be obtained by compiling its semantic equivalence. For example, in order to compile $\mathsf{Before}^- \ \phi' : \phi''$, I could instead compile $\neg\mathsf{Before}^+ \ \phi' : \neg\phi''$, which is a semantic equivalence to $\mathsf{Before}^- \ \phi' : \phi''$, but with only primitive operators. However, in this algorithm, we provide compilations for both the primitive and the derived operators. Such a direct compilation is more straightforward.

Let us go through the compilation algorithm case by case.

Case ψ. By Definition 7, ψ is fulfilled iff the first action of a trace satisfies the action formula ψ. Therefore, the recognition of a trace happens only at the initial state: i.e., for a trace, of which the first action satisfies ϕ, the OM transitions to an empty state. Otherwise, the OM transitions to an non-empty state, and is trapped there. The implication is that ψ is violated by the trace. Formally, the compilation is as follows.

$$\texttt{compile}(\psi) = \langle \boldsymbol{mon}, 1, \mathcal{M}, \{o_1\} \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_1, o_2, o_r\}, \iota, \{o_1, o_2\} \rangle$$

such that

$$\begin{aligned} \iota(o_1) &= \langle \psi, \emptyset, \{o_1, o_2\} \rangle \\ \iota(o_2) &= \langle \texttt{!}\ \psi, \{o_r\}, \{o_1, o_2\} \rangle \\ \iota(o_r) &= \langle \texttt{true}, \emptyset, \emptyset \rangle \end{aligned}$$

At the initial state, if the input action a is such that $a \Vdash \psi$, then o_1 is fired. This removes both o_1 and o_2 from the state, and puts $\mathcal{M}$ into an empty state. Otherwise, $a \nVdash \psi$, and thus o_2 is fired. This removes o_1 and o_2 from the state, while adding back o_r. As a result, $\mathcal{M}$ is trapped at a non-empty state.

If $\psi = \texttt{true} \equiv \langle \texttt{1} \rangle$, then a simpler OM can be generated.

$$\mathcal{M}_{\langle \texttt{1} \rangle} = \langle \mathcal{L}_\Sigma, \{o_1\}, \iota, \{o_1\} \rangle,$$

where

$$\iota(o_1) = \langle \texttt{true}, \emptyset, \{o_1\}\rangle$$

By the compilation of ψ, o_2 shall be a trigger rule $\langle \texttt{!}\ \texttt{true}, \{o_r\}, \{o_1, o_2\}\rangle$, which is never fired, and thus o_r is never added to the monitor state of $\mathcal{M}_{\langle 1\rangle}$. So both o_2 and o_r could be omitted in building $\mathcal{M}_{\langle 1\rangle}$. As a result, for a non-empty trace, o_1 is always fired at the initial state of $\mathcal{M}_{\langle 1\rangle}$, which puts the OM to an empty state.

Case $[\psi]$**.** Symmetric to the case ψ, $[\psi]$ could be violated iff the first action of a trace fails to satisfy the action formula ψ. Therefore, the rejection of a trace happens only at the initial state: i.e., for a trace, of which the first action satisfies ψ, the trace is accepted. Otherwise, it is rejected. Therefore, the compilation is as follows.

$$\texttt{compile}([\psi]) = \langle \boldsymbol{enf}, 1, \mathcal{M}, \emptyset\rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_1, o_2\}, \iota, \{o_1, o_2\}\rangle$$

such that

$$\begin{aligned}\iota(o_1) &= \psi,\\ \iota(o_2) &= \langle \psi, \emptyset, \{o_1, o_2\}\rangle.\end{aligned}$$

At the initial state, if the input action a is such that $a \Vdash \psi$, it is accepted. Meanwhile, o_2 is fired, which removes both o_1 and o_2 from the state, and puts $\mathcal{M}$ to an empty state. Then all subsequent actions in the rest of the trace will be accepted by $\mathcal{M}$. Otherwise, $a \nVdash \psi$, so the trace is rejected.

Specifically, $\mathcal{M}$ for $[\texttt{true}]$ is merely an empty machine as follows.

$$\mathcal{M}_{[\texttt{true}]} = \langle \mathcal{L}_\Sigma, \emptyset, \emptyset, \emptyset\rangle$$

Because there is no simple condition and trigger rule in the OM, all traces are accepted.

Case $\neg\phi'$**.** The compilation for $\neg\phi'$ depends on the the compilation for its sub-formula ϕ', and how ϕ' is typed. Suppose $\mathcal{M}$ and $\mathcal{M}'$ are the OMs for $\neg\phi'$ and ϕ' respectively. Then if ϕ' is typed as $\boldsymbol{enf}$, meaning that $\neg\phi'$ is typed as $\boldsymbol{mon}$, $\mathcal{M}$ recognizes a trace iff $\mathcal{M}'$ rejects the trace. The recognition of a trace in $\mathcal{M}$ shall coincide with the rejection of the same trace in $\mathcal{M}'$. Conversely, if ϕ' is typed as $\boldsymbol{mon}$, meaning that $\neg\phi'$ is typed as $\boldsymbol{enf}$, $\mathcal{M}$ rejects a trace iff $\mathcal{M}'$ recognizes the trace. As well, the rejection of a trace in $\mathcal{M}$ coincides with the recognition of the same trace in $\mathcal{M}'$.

Given the two scenarios described above, the compilation for $\neg\phi'$ is built in the following way.

1. Suppose $\langle \boldsymbol{enf}, k, \mathcal{M}', \emptyset\rangle$ is a compilation for ϕ', where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S_0'\rangle$. Then the compilation for $\neg\phi'$ is as follows:

$$\texttt{compile}(\neg\phi') = \langle \boldsymbol{mon}, k, \mathcal{M}, \mathsf{cond}_{\mathcal{M}'}(\mathcal{O}')\rangle$$

   where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \{o^*\}, \iota, S_0' \uplus \{o^*\}\rangle$$

   such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del}\rangle \\ \langle \texttt{!}\ \psi, \emptyset, \mathcal{O}' \uplus \{o^*\}\rangle & o \in \mathcal{O}' \wedge \iota'(o) = \psi \\ \langle \texttt{true}, \emptyset, \emptyset\rangle & o = o^* \end{cases}$$

Note that $\mathcal{M}$ is built from $\mathcal{M}'$ by converting each simple condition in $\mathcal{M}'$ to a final trigger rule in such a way that the trigger condition of the rule is merely the boolean negation of the original simple condition. Consequently, the action a in a trace that originally violates a simple condition at a monitor state of $\mathcal{M}'$, will instead fire the corresponding final trigger rule in $\mathcal{M}$, and put $\mathcal{M}$ to an empty state (if the same trace is fed into $\mathcal{M}$).

Note, however, $\mathcal{M}'$ as an enforcer could also enter an empty state by accepting a trace (e.g., an OM enforcing a k-bounded safety property might enter an empty state after k steps). It means that $\mathcal{M}'$ stops monitoring because no violation will be detected after the empty state. In this scenario, $\mathcal{M}$ shall enter a non-empty state, and get trapped. Therefore, $\mathcal{M}$ has o^* in its initial state. The obligation identifier o^* corresponds to such a trigger rule that it is always fired but posts no updates to the monitor states, and it is removed from the state iff some final trigger rule is fired. As a result, given a trace, which is accepted by $\mathcal{M}'$ and makes $\mathcal{M}'$ transition to an empty state, the trace is never recognized by $\mathcal{M}$.

2. Suppose $\langle \boldsymbol{mon}, k, \mathcal{M}', \mathcal{O}'_f \rangle$ is a compilation for ϕ', where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$. Then the compilation for $\neg\phi'$ is as follows:

$$\texttt{compile}(\neg\phi') = \langle \boldsymbol{enf}, k, \mathcal{M}, \emptyset \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota, S'_0 \rangle$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \texttt{!}\ \psi & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

Symmetric to the previous compilation, $\mathcal{M}$ is built from $\mathcal{M}'$ by converting each final trigger rule to a simple condition in such a way that the simple condition is merely the boolean negation of the final trigger rule's trigger condition. Therefore, an action a in some trace that originally fires a final trigger rules at some monitor state of $\mathcal{M}'$, will instead violates the corresponding simple condition in $\mathcal{M}$.

Case $\phi' \wedge \phi''$. The OM $\mathcal{M}$ of $\phi' \wedge \phi''$ is built from the compilations of its sub-formulae ϕ' and ϕ'', and depends on how they are typed. Suppose $\mathcal{M}'$ and $\mathcal{M}''$ are the compilations for ϕ' and ϕ''. Then intuitively, $\mathcal{M}$ runs as running $\mathcal{M}'$ and $\mathcal{M}''$ in parallel: i.e., if ϕ' and ϕ'' are typed as $\boldsymbol{enf}$, $\mathcal{M}$ accepts a trace iff the trace is accepted by both $\mathcal{M}'$ and $\mathcal{M}''$. Otherwise, if ϕ' and ϕ'' are typed as $\boldsymbol{mon}$, $\mathcal{M}$ recognizes a trace iff the trace is recognized by both $\mathcal{M}'$ and $\mathcal{M}''$. More formally,

1. Suppose $\langle \boldsymbol{enf}, k', \mathcal{M}', \emptyset \rangle$ and $\langle \boldsymbol{enf}, k'', \mathcal{M}'', \emptyset \rangle$ are the compilations for ϕ' and ϕ'' respectively, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. Then the compilation for $\phi' \wedge \phi''$ is given below:

$$\texttt{compile}(\phi' \wedge \phi'') = \langle \boldsymbol{enf}, \boldsymbol{max}\left(k',\ k''\right), \mathcal{M}, \emptyset \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \uplus S''_0 \rangle$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \\ \iota''(o) & o \in \mathcal{O}'' \end{cases}$$

The interpretation is rather straightforward: the action in a trace is rejected by either $\mathcal{M}'$ or $\mathcal{M}''$ must fail to satisfy some simple condition $o \in \mathcal{O}' \uplus \mathcal{O}''$, hence is also rejected by $\mathcal{M}$.

2. The compilation for this the case when ϕ' and ϕ'' are typed as $\boldsymbol{mon}$ is not as straightforward as the previous one. The difficulty is to identify the set of final trigger rules for $\mathcal{M}$. Suppose ϕ' discharges first, the final trigger rules for $\mathcal{M}''$ behaves as the final trigger rules for $\mathcal{M}$, but if ϕ'' discharges first, the final trigger rules $\mathcal{M}'$ will instead function as the final trigger rules for $\mathcal{M}$.

   The algorithm to identify the set of final trigger rules is based on Proposition 25 in Section 5. The procedure is described in the proof of the lemma. More formally, suppose $\langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f\rangle$ and $\langle \boldsymbol{mon}, k'', \mathcal{M}'', \mathcal{O}''_f\rangle$ are the compilations for ϕ' and ϕ'' respectively, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0\rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0\rangle$. Then the compilation for $\phi' \wedge \phi''$ is given below:

   $$\texttt{compile}(\phi' \wedge \phi'') = \langle \boldsymbol{mon}, \boldsymbol{max}\left(k',\ k''\right), \mathcal{M}, \mathcal{O}_f\rangle$$

   where

   $$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0\rangle$$

   To come up with $\mathcal{M}$ and $\mathcal{O}_f$, we need to use an intermediate OM $\mathcal{M}^*$, which is structured as follows:

   $$\mathcal{M}^* = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota^*, S'_0 \uplus S''_0\rangle$$

   such that

   $$\iota^*(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \\ \iota''(o) & o \in \mathcal{O}'' \end{cases}$$

   It is not hard to tell that the following connection exists between $\mathcal{M}^*$ and $\mathcal{M}'$ and $\mathcal{M}''$. Suppose $S' \in \mathsf{states}_{\mathcal{M}'}$ and $S'' \in \mathsf{states}_{\mathcal{M}''}$, then

   $$\exists w \in \Sigma^*.\, S'_0 \xrightarrow{w}_{\mathcal{M}'} S'_1.\, S''_0 \xrightarrow{w}_{\mathcal{M}''} S''_1 \Rightarrow$$
   $$\exists S^* \in \mathsf{states}_{\mathcal{M}}.\, \left(S^*_0 = S'_0 \uplus S''_0 \wedge S^*_1 = S'_1 \uplus S''_1 \wedge S^*_0 \xrightarrow{w}_{\mathcal{M}} S^*_1\right)$$

   Note that $\mathcal{M}^*$ is a simple "union" of $\mathcal{M}'$ and $\mathcal{M}''$: i.e., $\mathcal{M}'$ and $\mathcal{M}''$ are running independently at the same time, and the state transition of one OM does not affect another OM. So it is obvious that $\mathcal{M}^*$ monitors $\phi_1 \wedge \phi_2$. According to Proposition 25, as $\mathcal{M}^*$ is a pure OM, $\mathcal{M}^*$ can be converted to a pure OM $\mathcal{M}$ with a set of final trigger rules $\mathcal{O}_f$ following the procedure described in the proof of Proposition 25.

Case $\phi' \vee \phi'$. The compilation for $\phi' \vee \phi''$ is symmetrical to the one for $\phi' \wedge \phi''$. The intuitive compilation is also running two OMs $\mathcal{M}'$ and $\mathcal{M}''$ (for ϕ' and ϕ'' respectively) in parallel: i.e., suppose $\mathcal{M}$ is the OM for $\phi' \vee \phi''$, if both ϕ' and ϕ'' are typed as $\boldsymbol{mon}$, a trace is recognized by $\mathcal{M}$ iff it is recognized by either $\mathcal{M}'$ or $\mathcal{M}''$. Otherwise, if both ϕ' and ϕ'' are typed as $\boldsymbol{enf}$, a trace is accepted by $\mathcal{M}$ iff it is accepted by either $\mathcal{M}'$ or $\mathcal{M}''$. More formally,

1. Suppose $\langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f\rangle$ and $\langle \boldsymbol{mon}, k'', \mathcal{M}'', \mathcal{O}''_f\rangle$ are the compilations for ϕ' and ϕ'', where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0\rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0\rangle$. Then the compilation for $\phi' \vee \phi''$ is as follows:

   $$\texttt{compile}(\phi' \vee \phi'') = \langle \boldsymbol{mon}, \boldsymbol{max}\left(k',\ k''\right), \mathcal{M}, \mathcal{O}'_f \uplus \mathcal{O}''_f\rangle$$

   where

   $$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \uplus S''_0\rangle$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \backslash \mathcal{O}''_f \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \uplus \mathcal{O}'' \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}' \uplus \mathcal{O}_{del} \rangle & o \in \mathcal{O}''_f \wedge \iota''(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

$\mathcal{M}$ is built from $\mathcal{M}'$ and $\mathcal{M}''$ in such a way that each trigger condition in $\mathcal{M}'$ and $\mathcal{M}''$ is revised to "erase" both OMs. Consequently, if a trace that is recognized by either $\mathcal{M}'$ or $\mathcal{M}''$, a final trigger rule $o \in \mathcal{O}'_f \uplus \mathcal{O}''_f$ is fired, which puts both OMs to an empty state.

2. For the case when both ϕ' and ϕ'' are typed ***enf***, a trace is rejected iff both $\mathcal{M}'$ and $\mathcal{M}''$ rejects the trace. This cannot be captured by unifying $\mathcal{M}'$ and $\mathcal{M}''$ (as what we did for the previous sub-case). Therefore, we compile $\phi' \vee \phi''$ by compiling its semantical equivalence $\neg(\neg\phi' \wedge \neg\phi')$ instead.

   Suppose $\langle \textbf{\textit{enf}}, k', \mathcal{M}', \emptyset \rangle$ and $\langle \textbf{\textit{enf}}, k'', \mathcal{M}'', \emptyset \rangle$ are the compilations for ϕ' and ϕ'' respectively, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. Then the OM $\mathcal{M}$ for $\phi' \vee \phi''$ is constructed by the following three steps:

   (a) compile $\mathcal{M}'$ and $\mathcal{M}''$ to $\mathcal{M}'^*$ and $\mathcal{M}''^*$, where $\mathcal{M}'^*$ and $\mathcal{M}'^*$ are the OMs returned by `compile`$(\neg\phi')$ and `compile`$(\neg\phi'')$ respectively;

   (b) then compile $\mathcal{M}'^*$ and $\mathcal{M}''^*$ to $\mathcal{M}^*$ through `compile`$(\neg\phi' \wedge \neg\phi'')$;

   (c) lastly, compile $\mathcal{M}^*$ to $\mathcal{M}$ by `compile`$(\neg(\neg\phi' \wedge \neg\phi''))$.

Case Always ϕ'**.** Suppose $\langle \textbf{\textit{enf}}, k', \mathcal{M}', \emptyset \rangle$ is the compilation for ϕ', where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$.

Semantically, Always ϕ' imposes ϕ' on every suffix of a trace. So the intuitive compilation of Always ϕ' would be: for each step of transition, a new $\mathcal{M}'$ is started to enforce ϕ' on the suffix of a trace.

Suppose $\mathcal{M}'_1$, $\mathcal{M}'_2$, $\cdots$, $\mathcal{M}'_{k'+1}$ are $k'+1$ copies of $\mathcal{M}'$, where for each $i \in \{1, 2, \cdots, k'+1\}$, $\mathcal{M}'_i = \langle \mathcal{L}_\Sigma, \mathcal{O}'_i, \iota'_i, S'^i_0 \rangle$. Note that given i, $j \in \{1, 2, \cdots, k'+1\}$, the following statement holds for $\mathcal{M}'_i$ and $\mathcal{M}'_j$:

$$\mathcal{O}'_i \cap \mathcal{O}'_j = \emptyset \text{ iff } i \neq j.$$

The compilation for Always ϕ' is given below:

$$\texttt{compile}(\textsf{Always } \phi') = \langle \textbf{\textit{enf}}, \omega, \mathcal{M}, \emptyset \rangle$$

where

$$\begin{aligned} \mathcal{M} &= \langle \mathcal{L}_\Sigma, \mathcal{O}'_1 \uplus \mathcal{O}'_2 \uplus \cdots \uplus \mathcal{O}'_{k'+1} \uplus \mathcal{O}^*, \iota, S'^1_0 \uplus \{o^*_1\} \rangle \\ \mathcal{O}^* &= \{o^*_i \mid i = 1, 2, \cdots, k'+1\} \qquad (\mathcal{O}^* \cap \mathcal{O}'_i = \emptyset) \end{aligned}$$

such that

$$\iota(o) = \begin{cases} \iota'_i(o) & o \in \mathcal{O}'_i \\ \langle \texttt{true}, S'^{i+1}_0 \uplus \{o^*_{i+1}\}, (\mathcal{O}'_{i+1} \backslash S'^{i+1}_0) \uplus \{o^*_i\} \rangle & o \in \mathcal{O}^* \wedge o = o^*_i \wedge 1 \leq i \leq k' \\ \langle \texttt{true}, S'^1_0 \uplus \{o^*_1\}, (\mathcal{O}'_1 \backslash S'^1_0) \uplus \{o^*_{k'+1}\} \rangle & o \in \mathcal{O}^* \wedge o = o^*_{k'+1} \end{cases}$$

Note that $o^*_i \in \mathcal{O}^*$ is a trigger rule that is always fired. It "erases" the old copy of $\mathcal{M}'_j$ where $j = (i \ \textsf{mod}\ (k'+1)) + 1$, and starts a new instance of $\mathcal{M}'_j$. For example, by the $(k'+1)^{th}$ transition,

$o^*_{k'+1}$ is fired. This removes the obligation identifiers belongs to $\mathcal{M}'_1$ (the old copy of $\mathcal{M}'_1$ starting from the initial state) left in the monitor state, while adding back S^1_0 and o^*_1, which restarts a new copy of $\mathcal{M}'_1$.

This compilation works with $k' \in \mathbb{N}$, although the extension to $k' = \omega$ is straightforward, which involves infinite copies of $\mathcal{M}'$ and infinite number of o^*, it is in fact impractical when implemented due to the limitation of resources (e.g., memory). Therefore, our compilation requires the sub-formula ϕ' to be bounded by a finite bound.

Case $\mathsf{Eventually}\ \phi'$**.** As a dual of $\mathsf{Always}\ \phi'$, the compilation for $\mathsf{Eventually}\ \phi'$ follows exactly the same strategy as the previous one.

Suppose $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$, and $\mathcal{M}'_1$, $\mathcal{M}'_2$, $\cdots$, $\mathcal{M}'_{k+1}$ are $k'+1$ copies of $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$, where for each $i \in \{1, 2, \cdots, k'+1\}$, $\mathcal{M}'_i = \langle \mathcal{L}_\Sigma, \mathcal{O}'_i, \iota'_i, S'^i_0 \rangle$. Further suppose for each $i,\ j \in \{1, 2, \cdots, k'+1\}$, the following statement holds of $\mathcal{M}'_i$ and $\mathcal{M}'_j$:

$$\mathcal{O}'_i \cap \mathcal{O}'_i = \emptyset \text{ iff } i \neq j$$

Then the compilation for $\mathsf{Eventually}\ \phi'$ is given below:

$$\texttt{compile}(\mathsf{Eventually}\ \phi') = \langle \boldsymbol{mon}, \omega, \mathcal{M}, \mathcal{O}'^1_f \uplus \mathcal{O}'^2_f \uplus \cdots \mathcal{O}'^{k'+1}_f \rangle$$

where

$$\begin{aligned} \mathcal{M} &= \langle \mathcal{L}_\Sigma, \mathcal{O}'_1 \uplus \mathcal{O}'_2 \uplus \cdots \uplus \mathcal{O}'_{k+1} \uplus \mathcal{O}^*, \iota, S'^1_0 \uplus \{o^*_1\} \rangle \\ \mathcal{O}^* &= \{o^*_i \mid i = 1, 2, \cdots, k'+1\} \qquad (\mathcal{O}^* \cap \mathcal{O}'_i = \emptyset) \end{aligned}$$

such that

$$\iota(o) = \begin{cases} \iota'_i(o) & o \in \mathcal{O}'_i \backslash \mathcal{O}'^i_f \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}'_1 \uplus \mathcal{O}'_2 \uplus \cdots \uplus \mathcal{O}'_{k'+1} \uplus \mathcal{O}^* \rangle & o \in \mathcal{O}'^i_f \wedge \iota'_i(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle \texttt{true}, S'^{i+1}_0 \uplus \{o^*_{i+1}\}, (\mathcal{O}'_{i+1} \backslash S'^{i+1}_0) \uplus \{o^*_i\} \rangle & o \in \mathcal{O}^* \wedge o = o^*_i \wedge 1 \leq i \leq k' \\ \langle \texttt{true}, S'^1_0 \uplus \{o^*_1\}, (\mathcal{O}'_1 \backslash S'^1_0) \uplus \{o^*_{k'+1}\} \rangle & o \in \mathcal{O}^* \wedge o = o^*_{k'+1} \end{cases}$$

The trigger rule $o^*_i \in \mathcal{O}^*$ is always fired. It terminates the old copy of $\mathcal{M}'_j$ where $j = (i \ \mathsf{mod}\ (k'+1)) + 1$, and starts a new $\mathcal{M}'_j$. Such a compilation also requires the sub-formula ϕ' to be bounded by a finite bound.

Case $\mathsf{After}^+\ \phi' : \phi''$**.** By Definition 7, $\mathsf{After}^+\ \phi' : \phi''$ enforces ϕ'' after ϕ' is fulfilled. So the intuitive compilation for $\mathsf{After}^+\ \phi' : \phi''$ is to run $\mathcal{M}'$ and $\mathcal{M}''$ in sequence, where $\mathcal{M}'$ and $\mathcal{M}''$ are the OMs for ϕ' and ϕ'' respectively. As soon as $\mathcal{M}'$ detects the fulfillment of ϕ', it terminates itself and starts $\mathcal{M}''$.

Suppose $\langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$ and $\langle \boldsymbol{enf}, k', \mathcal{M}'', \emptyset \rangle$ are the compilations for ϕ' and ϕ'' respectively, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. Then, the compilation for $\mathsf{After}^+\ \phi' : \phi''$ is given below:

$$\texttt{compile}(\mathsf{After}^+\ \phi' : \phi'') = \langle \boldsymbol{enf}, k' + k'', \mathcal{M}, \emptyset \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \rangle$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \\ \langle \psi, S''_0, \mathcal{O}_{del} \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

First, $\mathcal{M}$ is initialized with S'_0 (the initial state of $\mathcal{M}'$), so it starts as $\mathcal{M}'$. Once the fulfillment of ϕ' is detected by $\mathcal{M}$, some trigger rule $o' \in \mathcal{O}'_f$ is triggered. This erases the state information of $\mathcal{M}'$, while adding back S''_0 (the initial state of $\mathcal{M}''$) to the monitor state. Thus, $\mathcal{M}$ stops $\mathcal{M}'$ and starts $\mathcal{M}''$.

Case $\mathsf{After}^-\ \phi' : \phi''$. Due to the duality of After^+ and After^-, compiling $\mathsf{After}^-\ \phi' : \phi''$ follows exactly the same strategy.

Supposing $\langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$ and $\langle \boldsymbol{mon}, k'', \mathcal{M}'', \mathcal{O}''_f \rangle$ are the compilations for ϕ' and ϕ'', where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. Then the OM for $\mathsf{After}^-\ \phi' : \phi''$ is given below:

$$\texttt{compile}(\mathsf{After}^-\ \phi' : \phi'') = \langle \boldsymbol{mon}, k' + k'', \mathcal{M}, \mathcal{O}''_f \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \rangle$$

in which

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \\ \langle \psi, S''_0, \mathcal{O}_{del} \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

Similar to the previous case, this OM is also initialized with S'_0 (the initial state of $\mathcal{M}'$), so it starts also as $\mathcal{M}'$. Once the fulfillment of ϕ' is detected by $\mathcal{M}$, some trigger rule $o' \in \mathcal{O}'_f$ is fired, which stops $\mathcal{M}'$ and starts $\mathcal{M}''$.

Case $\mathsf{Before}^+\ \phi' : \phi''$. By Definition 7, $\mathsf{Before}^+\ \phi' : \phi''$ enforces ϕ'' before ϕ' is fulfilled. So the intuitive compilation involves the running of $\mathcal{M}'$ and $\mathcal{M}''$ in parallel, where $\mathcal{M}'$ and $\mathcal{M}''$ are the OMs for the sub-formulae ϕ' and ϕ'' respectively. As soon as $\mathcal{M}'$ detects the fulfillment of ϕ', $\mathcal{M}'$ terminates $\mathcal{M}''$ by "erasing" the state information of the latter.

Suppose $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$ and $\texttt{compile}(\phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}'', \emptyset \rangle$, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. The compilation for $\mathsf{Before}^+\ \phi' : \phi''$ is as follows:

$$\texttt{compile}(\mathsf{Before}^+\ \phi' : \phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}, \emptyset \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \uplus S''_0 \rangle$$

in which

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \uplus \mathcal{O}'' \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

First, $\mathcal{M}$ is initialized with $S'_0 \uplus S''_0$ (the initial states of $\mathcal{M}'$ and $\mathcal{M}''$), so it starts $\mathcal{M}'$ and $\mathcal{M}''$ simultaneously. Suppose a trace fulfills ϕ', but does not violate ϕ'' up to the point of fulfillment of ϕ'. The trace will fires some trigger rule $o' \in \mathcal{O}_f$, which removes $\mathcal{O}''$ from the state, and transitions $\mathcal{M}'$ to an empty state. As a result, $\mathcal{M}''$ stops enforcing ϕ''.

Case $\mathsf{Before}^{-}\ \phi' : \phi''$**.** Although Before^{-} and Before^{+} are duals of each other, the compilation of $\mathsf{Before}^{-}\ \phi' : \phi''$ is slightly different from the one for the latter.

Semantically, $\mathsf{Before}^{-}\ \phi' : \phi''$ requires ϕ'' to be fulfilled no later than ϕ' is fulfilled. As well, the OM $\mathcal{M}$ for $\mathsf{Before}^{-}\ \phi' : \phi''$ runs $\mathcal{M}'$ and $\mathcal{M}''$ (for ϕ' and ϕ'' respectively) in parallel: i.e., if $\mathcal{M}'$ detects the fulfillment of ϕ' before $\mathcal{M}''$ does, which means $\mathsf{Before}^{-}\ \phi' : \phi''$ is violated, then $\mathcal{M}$ shall be trapped in an non-empty state. Otherwise $\mathcal{M}''$ detects the fulfillment of ϕ'' before or at the same time when $\mathcal{M}'$ does, $\mathcal{M}$ shall enter an empty state.

Suppose $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$ and $\texttt{compile}(\phi'') = \langle \boldsymbol{mon}, k'', \mathcal{M}'', \mathcal{O}''_f \rangle$, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$. Then the compilation for $\mathsf{Before}^{-}\ \phi' : \phi''$ is given below:

$$\texttt{compile}(\mathsf{Before}^{-}\ \phi' : \phi'') = \langle \boldsymbol{mon}, k'', \mathcal{M}, \mathcal{O}''_f \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'' \uplus \{o^*\}, \iota, S'_0 \uplus S''_0 \uplus \{o^*\} \rangle$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \backslash \mathcal{O}''_f \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \uplus \mathcal{O}'' \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}' \uplus \mathcal{O}_{del} \uplus \{o^*\} \rangle & o \in \mathcal{O}''_f \wedge \iota''(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle \texttt{true}, \emptyset, \emptyset \rangle & o = o^* \end{cases}$$

At first, $\mathcal{M}$ is initialized with $S'_0 \uplus S''_0 \uplus \{o^*\}$, so it starts $\mathcal{M}'$ and $\mathcal{M}''$ simultaneously. Suppose a trace fulfills ϕ' yet not fulfills ϕ'', a final trigger rule $o \in \mathcal{O}'_f$ is fired. This clears $\mathcal{M}''$ by removing its whole set of obligation identifiers $\mathcal{O}''$, while erasing the state information of $\mathcal{M}'$. As a result, $\mathcal{M}$ is trapped in an non-empty state containing only o^*. But if the trace fulfills ϕ'' up to the point of fulfillment of ϕ', a final trigger rule $o \in \mathcal{O}''_f$ is fired, which clears $\mathcal{M}'$ by removing its whole set of obligation identifiers $\mathcal{O}'$, erases the state information of $\mathcal{M}''$ and removes o^*. Consequently, it puts $\mathcal{M}$ to an empty state.

Case $\mathsf{Fulfilling}^{k'}\ \phi' \,?\, \phi'' : \phi'''$**.** Semantically, $\mathsf{Fulfilling}^{k'}\ \phi' \,?\, \phi'' : \phi'''$ enforces ϕ'' after the fulfillment of ϕ', or enforces ϕ''' if ϕ' is not fulfilled after k' steps. Suppose $\mathcal{M}'$, $\mathcal{M}''$ and $\mathcal{M}'''$ are the OMs for the sub-formulae ϕ', ϕ'' and ϕ''' respectively. Then the OM $\mathcal{M}$ for enforcing $\mathsf{Fulfilling}^{k'}\ \phi' \,?\, \phi'' : \phi'''$ behaves in the following way. First, it keeps a counter which counts the number of transitions $\mathcal{M}$ have made. If $\mathcal{M}'$ detects the fulfillment of ϕ' before the counter reaches k', it terminates $\mathcal{M}'$ and starts $\mathcal{M}''$. Otherwise, if up to the point when the counter reaches k', $\mathcal{M}'$ has not detected the fulfillment of ϕ' yet, $\mathcal{M}'$ is terminated as well, but $\mathcal{M}'''$ is started instead of $\mathcal{M}''$.

Suppose Further $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$, $\texttt{compile}(\phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}'', \emptyset \rangle$ and $\texttt{compile}(\phi''') = \langle \boldsymbol{enf}, k''', \mathcal{M}''', \emptyset \rangle$, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$, $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0 \rangle$ and $\mathcal{M}''' = \langle \mathcal{L}_\Sigma, \mathcal{O}''', \iota''', S'''_0 \rangle$. Then,

$$\texttt{compile}(\mathsf{Fulfilling}^{k}\ \phi' \,?\, \phi'' : \phi''') = \langle \boldsymbol{enf}, k' + \boldsymbol{max}\left(k'',\ k'''\right), \mathcal{M}, \emptyset \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'' \uplus \mathcal{O}''' \uplus \mathcal{O}^*, \iota, S'_0 \uplus \{o^*_1\} \rangle$$
$$\mathcal{O}^* = \{o^*_i \mid i = 1, 2, \cdots, k\}$$

such that

$$\iota(o) = \begin{cases} \iota'(o) & o \in \mathcal{O}' \backslash \mathcal{O}'_f \\ \iota''(o) & o \in \mathcal{O}'' \\ \iota'''(o) & o \in \mathcal{O}''' \\ \langle \psi, S''_0, \mathcal{O}_{del} \uplus \mathcal{O}^* \uplus S'''_0 \rangle & o \in \mathcal{O}'_f \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \\ \langle \texttt{true}, \{o^*_{i+1}\}, \{o^*_i\} \rangle & o = o^*_i \wedge 1 \le i < k' \\ \langle \texttt{true}, S'''_0, \mathcal{O}' \uplus \{o^*_{k'}\} \rangle & o = o^*_i \wedge i = k' \end{cases}$$

Note that the set $\mathcal{O}^*$ of trigger rules are used to count the number of transitions made by $\mathcal{M}$.

At first, $\mathcal{M}$ is initialized with $S'_0 \uplus \{o^*_1\}$, where S'_0 starts $\mathcal{M}'$ and o^*_1 starts the counters. The counters work in the following way. By the i^{th} ($i < k$) transition, $o^*_i \in \mathcal{O}^*$ is always fired, which removes o^*_i from the monitor state, while adding back o^*_{i+1}. So they could remember the number of transitions that $\mathcal{M}'$ have made. At the k'^{th} transition, $o^*_{k'}$ is fired, which removes $o^*_{k'}$ from the state, puts $\mathcal{M}'$ to an empty state, and adds back S'''_0 (the initial state of $\mathcal{M}'''$) to start $\mathcal{M}'''$. The counters can be stopped if any $o \in \mathcal{O}'_f$ is fired, which removes the set of counters $\mathcal{O}^*$ from the state, puts $\mathcal{M}'$ to an empty state and stops $\mathcal{M}''$ if S''_0 is added. Note that if $o \in \mathcal{O}'_f$ is fired right at the k'^{th} transition, $\mathcal{M}'''$ will not be started. In addition, $o \in \mathcal{O}'_f$ is never fired after the k'^{th} transition.

Case Ignoring $\psi' : \phi'$. The compilation for this case is rather simple: i.e., any action satisfying ψ' shall be ignored by $\mathcal{M}'$ (the OM for ϕ'). That is, the action will not cause a transition or a violation.

Suppose $\texttt{compile}(\phi') = \langle \mathcal{T}, \mathcal{B}, \mathcal{M}', \mathcal{O}_f \rangle$, where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0 \rangle$. Then,

$$\texttt{compile}(\mathsf{Ignoring}\ \psi' : \phi') = \langle \mathcal{T}, \mathcal{B}, \mathcal{M}, \mathcal{O}_f \rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota, S'_0 \rangle$$

such that

$$\iota(o) = \begin{cases} \psi \texttt{ || } \psi' & o \in \mathcal{O}' \wedge \iota'(o) = \psi \\ \langle \psi \texttt{ \&\& ! } \psi', \mathcal{O}_{add}, \mathcal{O}_{del} \rangle & o \in \mathcal{O}' \wedge \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \end{cases}$$

Obviously, $\mathcal{M}$ is built from $\mathcal{M}'$ in such a way that each simple condition ψ is converted to $\psi \texttt{ || } \psi'$, and each trigger rule with trigger condition ψ is converted to a rule with the same $\mathcal{O}_{add}$ and $\mathcal{O}_{del}$, but different trigger condition $\psi \texttt{ \&\& ! } \psi'$. Therefore, for any action satisfying ψ', no simple condition will be violated and no trigger rule will be fired.

8.2 Examples

In this section, we provide some example compilations of several simple formulae.

Example 27. *Suppose we are to compile* Always $[q]$. *Further suppose* $\texttt{compile}([q]) = \langle \boldsymbol{enf}, 1, \mathcal{M}, \emptyset \rangle$ *where* $\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_1, o_2\}, \iota, \{o_1, o_2\} \rangle$ *such that* $\iota(o_1) = q$, *and* $\iota(o_2) = \langle q, \emptyset, \{o_1, o_2\} \rangle$. *Then*

$$\texttt{compile}(\mathsf{Always}\ [q]) = \langle \boldsymbol{enf}, \omega, \mathcal{M}', \emptyset \rangle$$

where

$$\mathcal{M}' = \langle \mathcal{L}_\Sigma, \{o_i \,|\, i = 1, 2, \ldots, 6\}, \iota', \{o_1, o_3, o_5\} \rangle$$

such that

$$\begin{aligned}
\iota'(o_1) &= q\\
\iota'(o_2) &= \langle q, \emptyset, \{o_1, o_3\}\rangle\\
\iota'(o_3) &= q\\
\iota'(o_4) &= \langle q, \emptyset, \{o_2, o_4\}\rangle\\
\iota'(o_5) &= \langle \texttt{true}, \{o_2, o_4, o_6\}, \{o_5\}\rangle\\
\iota'(o_6) &= \langle \texttt{true}, \{o_1, o_3, o_5\}, \{o_6\}\rangle
\end{aligned}$$

Example 28. *Suppose we are to compile* Eventually ψ*. Further suppose* $\texttt{compile}(\psi) = \langle \boldsymbol{mon}, 1, \mathcal{M}, \{o_1\}\rangle$*, where* $\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_1, o_2, o_r\}, \iota, \{o_1, o_2\}\rangle$ *such that* $\iota(o_r) = \langle\ \texttt{true}, \emptyset, \emptyset\rangle$*,* $\iota(o_1) = \langle \psi, \emptyset, \{o_1, o_2\}\rangle$*, and* $\iota(o_2) = \langle !\ \psi, \{o_r\}, \{o_1, o_2\}\rangle$*. Then,*

$$\texttt{compile}(\textsf{Eventually } \psi) = \langle \boldsymbol{mon}, \omega, \mathcal{M}', \{o_1, o_2\}\rangle$$

where

$$\mathcal{M}' = \langle \mathcal{L}_\Sigma, \{o_i \mid i = 1, 2, \ldots, 8\}, \iota', \{o_1, o_3, o_5\}\rangle$$

such that

$$\begin{aligned}
\iota'(o_1) &= \langle \psi, \emptyset, \{o_i \mid i = 1, 2, \ldots, 8\}\rangle\\
\iota'(o_2) &= \langle \psi, \emptyset, \{o_i \mid i = 1, 2, \ldots, 8\}\rangle\\
\iota'(o_3) &= \langle !\ \psi, \{o_7\}, \{o_1, o_3\}\rangle\\
\iota'(o_4) &= \langle !\ \psi, \{o_8\}, \{o_2, o_4\}\rangle\\
\iota'(o_5) &= \langle \texttt{true}, \{o_2, o_4, o_6\}, \{o_5, o_8\}\rangle\\
\iota'(o_6) &= \langle \texttt{true}, \{o_1, o_3, o_5\}, \{o_6, o_7\}\rangle\\
\iota'(o_7) &= \langle \texttt{true}, \emptyset, \emptyset\rangle\\
\iota'(o_8) &= \langle \texttt{true}, \emptyset, \emptyset\rangle
\end{aligned}$$

Example 29. *Let us take the results of Example (27) and (28) to compile* After$^+$ Eventually p : Always $[q]$ *as follows.*

$$\texttt{compile}(\textsf{After}^+ \textsf{ Eventually } p : \textsf{Always } q) = \langle \boldsymbol{enf}, \omega, \mathcal{M}, \emptyset\rangle$$

where

$$\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_i \mid i = 1, 2, \ldots, 14\}, \iota', \{o_1, o_3, o_5\}\rangle$$

such that

$$\begin{aligned}
\iota'(o_1) &= \langle \psi, \{o_9, o_{11}, o_{13}\}, \{o_i \,|\, i = 1, 2, \ldots, 8\}\rangle\\
\iota'(o_2) &= \langle \psi, \{o_9, o_{11}, o_{13}\}, \{o_i \,|\, i = 1, 2, \ldots, 8\}\rangle\\
\iota'(o_3) &= \langle \texttt{!}\ \psi, \{o_7\}, \{o_1, o_3\}\rangle\\
\iota'(o_4) &= \langle \texttt{!}\ \psi, \{o_8\}, \{o_2, o_4\}\rangle\\
\iota'(o_5) &= \langle \texttt{true}, \{o_2, o_4, o_6\}, \{o_5, o_8\}\rangle\\
\iota'(o_6) &= \langle \texttt{true}, \{o_1, o_3, o_5\}, \{o_6, o_7\}\rangle\\
\iota'(o_7) &= \langle \texttt{true}, \emptyset, \emptyset\rangle\\
\iota'(o_8) &= \langle \texttt{true}, \emptyset, \emptyset\rangle\\
\iota'(o_9) &= q\\
\iota'(o_{10}) &= \langle q, \emptyset, \{o_9, o_{11}\}\rangle\\
\iota'(o_{11}) &= q\\
\iota'(o_{12}) &= \langle q, \emptyset, \{o_{10}, o_{12}\}\rangle\\
\iota'(o_{13}) &= \langle \texttt{true}, \{o_{10}, o_{12}, o_{14}\}, \{o_{13}\}\rangle\\
\iota'(o_{14}) &= \langle \texttt{true}, \{o_9, o_{11}, o_{13}\}, \{o_{14}\}\rangle
\end{aligned}$$

8.3 Correctness of Compilation and Soundness of Typing

In this section, we present a theorem which demonstrates the correctness of the compilation algorithm, and the soundness of the type system.

Theorem 30. (Correctness Theorem) *Given any well-typed obligation policy ϕ, suppose* `compile`$(\phi) = \langle \mathcal{T}, \mathcal{B}, \mathcal{M}, \mathcal{O}_f\rangle$. *Then both of the following hold:*

*C1: if $\phi : k$-**enf**, then*

*C1.1: $\mathcal{T} =$ **enf**, $\mathcal{B} = k$, and $\phi = \mathcal{L}_{acc}(\mathcal{M})$;*

C1.2: if $k \in \mathbb{N}$, then $\mathcal{L}_{acc}(\mathcal{M})$ is a k-bounded safety property.

*C2: if $\phi : k$-**mon**, then*

*C2.1: $\mathcal{T} =$ **mon**, $\mathcal{B} = k$, $\phi = \mathcal{L}_{rec}(\mathcal{M})$, and $\mathcal{M}$ is a pure OM with final trigger rules $\mathcal{O}_f$;*

C2.2: if $k \in \mathbb{N}$, then $\mathcal{L}_{rec}(\mathcal{M})$ is a k-bounded guarantee property.

This theorem states that if a formula ϕ is typed as k-***enf*** (or k-***mon*** respectively), the OM returned from `compile`(ϕ) accepts (or recognizes respectively) only those traces that satisfy ϕ. As pointed out in Section 5, a property accepted (or recognized) by an OM is a non-empty safety property (or a non-universal guarantee property). Therefore, it is easy to conclude that if a formula ϕ is typed as k-***enf***, ϕ is a k-bounded safety property. An analogous conclusion is applicable to formulae typed as k-***mon***. Thus, the correctness theorem of the compilation also entails the soundness of the type system. A proof of this theorem can be found in Appendix A.

9 Related Work

This section examines some related efforts with respect to the specification and enforcement of obligation policies.

Specification of obligation policies. So far, many policy languages have been proposed for specifying obligations. Most of them, however, defined obligations as an extension to an access control framework, and provided only syntactic elements for expressing obligations, while leaving the semantics open to interpretation. Examples are: EPAL (Enterprise Privacy Authorization Language) [5] — a language for specifying enterprise privacy policies that adds the purpose of uses to the access decisions, XACML [23] — a modern access control language that associates obligations with decisions, and Ponder [14] — a language for policy-based management of network and distributed system that defined obligation policies explicitly in the form of event-triggered rules.

Hilty *et al.* [24] proposed the obligation specification language (OSL), specifically for expressing obligations. In their work, they carefully defined the syntax and semantics for the language, based on temporal logic. The enforcement of OSL policies have been studied in their technical report [26], which is achieved by translating OSL to ODRLc (a subset of ODRL [27]), and directly deploying the enforcement mechanisms for digital rights management (DRM). Compared to OSL, our policy language attempts to capture the idiomatic components of obligation policies in the framework of temporal logic.

Kagal *et al.* [29] proposed the language Rei based on deontic logic [36]. Rei is a general-purpose policy language, which not only specifies security policies, but also management policies and conversation policies. Such nature of the language determines that obligation policies will not be treated in detail in Rei. we described in Section 6 that the simple conditions and trigger rules of OM are sufficient for capturing the four deontic elements in Rei.

Similar to our language, there are also other temporal logics proposed to incorporate both finite and finite traces. Examples are finite-time LTL (FLTL) [31] by Lichtenstein *et al.*, LTL_3 [8] by Bauer *et al.*, RV-LTL [9] by Bauer *et al.* and $\text{LTL}^{\pm}$ by Eisner *et al.* [17, 16]. While these LTLs are tailored for evaluating finite traces, the syntactical classification of formulae is no longer obvious. Compared to these LTLs, our language captures patterns of policy composition that result in enforceable policies.

Semantically, formulae of the form $\mathsf{After}^{+}\ \phi_1 : \phi_2$ (or $\mathsf{After}^{-}\ \phi_1 : \phi_2$) defines a scope with the starting instant (i.e., the first instant ϕ_1 is fulfilled) on a trace to evaluate ϕ_2. This idea of scoping is similar to the N (Now) modality [30] introduced by Laroussine *et al.*, which defines a scope with fixed starting instant "now" to evaluate some property.

Implementability check. Schneider pioneered the study of the enforceability of security properties [43]. The discussion was based on the standard safety-liveness classification of properties [3, 2]. He showed that reference monitor can only enforce safety properties.

More recently, Ligatti *et al.* [32, 33, 34, 35] showed that it is possible to enforce at runtime more than safety properties, using edit automata (instead of simply halting the system, edit automata is able to suppress and insert actions into the current execution sequence). Although edit automata can enforce a large class of infinite renewal properties (which is capable of expressing certain obligations in security policies), the insertion and suppression operations of the edit automata model is sometimes beyond practicality when actually implemented.

A syntactic classification for standard LTL formulae based on the safety-liveness hierarchy has been presented by Sistla [44]. Compared to his approach, our type system offers the advantage of providing feedbacks to policy developers regarding which part of a policy needs refinement in order to guarantee enforceability.

Enforcement of obligation policies. Dougherty *et al.* [15] defined a rich model of obligations. They also identified that obligation policies contain both safety and guarantee properties, and their boolean combinations. The approach they proposed to enforce non-safety obligations is to use $\mathcal{S}$*-safety closure* [15] to approximate obligations.

Falcone *et al.* [19, 20, 21] presented a unified view of runtime verification and enforcement of properties in Streett automata [45] based on the safety-progress classification. They proposed a systematic approach to produce an enforcement monitor from a Streett automaton for the enforceable properties (i.e., safety, guarantee, response and obligation properties). However, their monitor model relies on the *store* and *dump* operations [19], which confronts the same dilemma as the edit automata model.

Ribeiro *et al.* [41] suggested the idea of enforcing obligation-based security policies using transactional rollback [12, 42] in a special execution environment, where several distinct actions can be interrelated in an atomic way. Gama and Ferreira designed a platform called Heimdall [22] for enforcing obligation-based policies. Their platform, however, requires that any executed actions of the system can be counterbalanced in the future. Obviously, the above two approaches for enforcing obligation policies are only applicable to certain systems.

Irwin *et al.* [28] suggested to enforce an obligation policy by detecting the accountability of the policy: i.e., whether at a state, an entity has enough privileges, or resources to perform the obligation. Later, Pontual *et al.* [40] extended the work on practices (e.g., system architecture, support tools) about detecting and resolving violations in terms of accountability.

Other works related to obligation policies. Except for those works mentioned above, which are related to the three stages of the work-flow (as shown in Figure 1), there are also other researches on the impact of adding obligations into the policy system. Bettini *et al.* [11] investigated the problem of choosing the best policy rules to minimize the provisions and obligations that a user has to fulfill based on the numerical weight assigned to them as well as their semantical relations. Backes *et al.* [7] presented a toolkit for EPAL [4] to detect refinement of EPAL policies (i.e., whether fulfilling one policy will automatically fulfill another policy). Similar work was also done by Ni *et al.* [38], where they used the term "dominance" instead of "refinement". Besides, the conflicts between obligations and permissions while adding obligations into the policy system are studied in [29, 38].

10 Conclusion

In this report, we have presented a tool-chain to support the development of obligation policies for program monitoring. This work streamlines the whole work-flow illustrated by Figure 1 in Section 1.

10.1 Contributions

First of all, we defined an obligation policy language, which is able to express the common idiomatic components of obligation policies. Formal semantics for the language have been defined. The use of the language was demonstrated by means of case studies.

Secondly, we defined a type system, which enable us to syntactically identify if an obligation policy (formulated in the obligation policy language) is enforceable or monitorable by a reference

[4]In fact, the toolkit was proposed on the basis of E-P3P [6], which is close to EPAL except for that it is using abstract syntax, to avoid the lengthy XML syntax used by EPAL.

monitor. This paves the way for translating an obligation policy to a representation of enforcement or monitoring mechanism. As well, the type system can be used as a feedback scheme in formulating an enforceable obligation policy. This use of type system has been demonstrated by a dedicated case study.

We also defined the model of obligation monitors, which represents an enforcement mechanism for safety properties, and a monitoring mechanism for guarantee properties. We sketched how obligation monitors can be used to capture the four deontic components in Rei [29], and why obligation monitors facilitate the inlined program monitoring.

Finally, we developed an algorithm that compiles a well-typed obligation policy to an obligation monitor. The correctness theorem for the algorithm has been proved. The correctness result also serves as the soundness proof of our type system.

10.2 Future Works

The work presented here suggests several directions for future work.

Enforcement of obligation policies in some extensible platform. The first direction of future research would be to implement the tool-chain in some extensible system, such as Android — a smart phone platform. To that end, there is still an important building block missing in the process, which is a framework for injecting obligation monitors to Android applications.

Automatic optimization of obligation monitors. Another promising direction to extend this work is to design an algorithm that will reduce the complexity of an OM. Such optimization can be achieved in different level as follows.

- Language-level: formulae in our language could be somehow simplified to reduce the complexity of OMs generated by the compiler. For example, the OMs compiled from Always Always $[p]$ and Always $[p]$ are tremendously different in complexity, although Always Always $[p]$ and Always $[p]$ are semantically equivalent.
- Model-level: the OMs returned from the compiler also leave some space for optimization. For example, trigger rules that will be fired by the same action could be further compressed to one rule.
- Implementation-level: as sketched in Section 6, the model of OM shares some commonalities with the policy representation adopted by Yan and Fong in [46]. Therefore, we anticipate that the optimization through constant propagation and liveness analysis used in their work is also applicable to the inlined implementation of OMs.

Theoretical properties of the policy language. It is interesting to explore the expressive power of this language compared to other temporal logics (e.g., LTL, FLTL, etc.). The translation between our language and other temporal logics can also be studied.

References

[1] Irem Aktug and Katsiaryna Nalluka. ConSpec - a formal language for policy specification. In *Proceedings of the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, Dresden, Germany, February 2007.

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

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[4] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Divisions (AFSC), October 1972.

[5] Paul Ashley, Satoshi Hada, Günter Karjoth ad Calvin Powers, and Matthias Schunter. Enterprise privacy authorization language (EPAL 1.2). `http://www.zurich.ibm.com/security/enterprise-privacy/epal`, 2003.

[6] Paul Ashley, Satoshi Hada, Günter Karjoth, and Matthias Schunter. E-P3P privacy policy and privacy authorization. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society (WPES'02)*, pages 103–109, Washington, DC, USA, November 2002.

[7] Michael Backes, Birgit Pfitzmann, and Matthias Schunter. A toolkit for managing enterprise privacy policies. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS'03)*, volume 2808 of *Lecture Notes in Computer Science*, pages 162–180, Norway, October 2003.

[8] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 4337, pages 260–272. Springer Berline / Heidelberg, 2006.

[9] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad and the ugly, but how ugly is ugly? In *Proceedings of the 7th International Conference on Runtime Verification (RV'07)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Vancouver, Canada, March 2007. Springer-Verlag.

[10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical report tum-i0724, Institut für Informatik, Technische Universität München, 2007.

[11] Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminada. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[12] Arnar Birgisson, Mohan Dhawan, Ulfar Erlingsson, Vinod Ganapathy, and Liviu Lftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM Conference on Computer and Communication Security (CCS'08)*, pages 223–234, Alexandria, Virginia, USA, October 2008.

[13] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specifications*, NATO Advanced Science Institutes Series, pages 143–202. Springer, 1991.

[14] Nicodemos Damianou, Daranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the 2001 IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, pages 18–38, London, UK, January 2001.

[15] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In J. Biskup and J. Lopes, editors, *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 375–389, Dresden, Germany, September 2007. Springer.

[16] Cindy Eisner, Dana Fisman, John Havlicek, Yaod Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39, Boulder, Colorado, USA, July 2003.

[17] Cindy Eisner, Dana Fisman, John Havlick, and Johan Mårtensson. The $\top$, $\bot$ approach for truncated semantics. Technical report, May 2006.

[18] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P'00)*, pages 246–255, Berkeley, California, USA, May 2000.

[19] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In R. Sekar and Arun Pujari, editors, *Information Systems Security*, volume 5352 of *Lecture Notes in Computer Science*, pages 41–55. Springer Berline / Heidelberg, 2008.

[20] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, number 8, pages 593–600, Honolulu, Hawaii, USA, March 2009.

[21] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron Peled, editors, *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer Berline / Heidelberg, 2009.

[22] Pedro Gama and Paulo Ferreira. Obligation policies: an enforcement platform. In *Proceedings of the 2005 IEEE Internation Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pages 203–212, Stockholm, Sweden, June 2005.

[23] Simon Godik and Tim Moses. eXtensible access control makeup language (XACML). Technical report, OASIS, May 2002.

[24] Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546, Dresden, Germany, September 2007.

[25] Manuel Hilty, Alexander Pretschner, Christian Schaefer, and Thomas Walter. A system model and an obligation specification language for distributed usage control. Technical report, DoCoMo Euro-lab, 2006.

[26] Manuel Hilty, Alexander Pretschner, Thomas Walter, and Christian Schaefer. Enforcement for usage control - an overview of control mechanisms. Technical report, DoCoMo Euro-labo, October 2006.

[27] Renato Iannella. Open digital rights language (odrl) version 1.1. `http://odrl.net/1.1/ODRL-11.pdf`, August 2002.

[28] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligation. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS'06)*, pages 134–143, Alexandria, Virginia, USA, October 2006.

[29] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 63–74, Lake Como, Italy, June 2003.

[30] François Laroussinie, Nicolas Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392, Washington, DC, USA, July 2002.

[31] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*. Springer Berline / Heidelberg, 1985.

[32] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time secuirty policies. *International Journal of Information Security*, 4(1-2):2–16, February 2005.

[33] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security properties with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373, Milan, Italy, September 2005.

[34] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security properties with program monitors. Technical Report TR-720-05, Princeton University, January 2005.

[35] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of non-safety policies. *ACM Transactions on Information and Systems Security*, 12(3), January 2009.

[36] John-Jules Ch. Meyer and Roel Wieringa. Deontic logic: a concise overview. In *Deontic Logic In Computer Science*, pages 3–16. John Wiley and Sons Ltd., Chichester, UK, 1993.

[37] Naftaly H. Minsky and Abe D. Lockman. Ensuring integrity by adding obligations to privilege. In *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, pages 92–102, London, England, 1985.

[38] Qun Ni, Elisa Bertino, and Jorge Lobo. An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM Symposium on Access Control and Technologies (SACMAT'08)*, pages 133–142, Estes Park, Colorado, USA, June 2008.

[39] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, New York, USA, September 1977.

[40] Murillo Pontual, Omar Chowdhury, William Winsborough, Ting Yu, and Keith Irwin. A framework for enforcing user obligations. In *proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT'11)*, Innsbruck, Austria, June 2011.

[41] Carlos Ribeiro, Andre Zuquete, and Paulo Ferreira. Enforcing obligation with security monitors. In *Third Internatonal Conference on Information and Communications Security (ICICS'01)*, pages 172–176, Xi'an, China, January 2001.

[42] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, pages 439–448, Washington, DC, USA, June 2002.

[43] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[44] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, September 1993.

[45] Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1-2):121–141, 1982.

[46] Fei Yan and Philip W. L. Fong. Efficient IRM enforcement of history-based access control policies. In *Proceedings of the 4th ACM Symposium on Information, Computer and Communications Security (ASIACCS'09)*, pages 35–46, Novotel Rockford Darling Harbour, Sydney, Australia, March 2009.

A Proof of Theorem 30

We provide in the following a formal proof to Theorem 30, which proceeds by a structural induction on the syntax of the formula ϕ (equivalently, the derivation tree of the typing of ϕ). The main idea is as follows. For each ϕ, suppose $\mathcal{M}$ is the OM returned from `compile`(ϕ). If ϕ is typed as k-***enf***, we prove that (a) a trace is rejected by $\mathcal{M}$ iff the trace violates ϕ, and (b) if $k \in \mathbb{N}$, and if $\mathcal{M}$ detects no violation within k steps, then $\mathcal{M}$ will stop monitoring. Conversely, if ϕ is typed as k-***mon***, we prove that (a) a trace is recognized by $\mathcal{M}$ iff the trace fulfills ϕ, and (b) $\mathcal{M}$ is a pure OM with a set of final trigger rules, firing of which coincides with the fulfillment of ϕ, and (c) if $k \in \mathbb{N}$, and if $\mathcal{M}$ does not enter an empty state within k steps, then $\mathcal{M}$ will get trapped at a non-empty state and stop monitoring.

Case ψ. Suppose $\mathcal{M}$ is the OM returned by `compile`(ψ) (refer to the corresponding compilation given in Section 8.1).

Let us prove that C2.1 holds for $\mathcal{M}$. By construction, $\mathcal{M}$ is a pure OM. According to Definition 7, given a trace $w \models \phi$, there exists $a \in \Sigma$ such that $a \preceq w$ and $a \Vdash \psi$. This trace will fire o_1, which puts $\mathcal{M}$ to an empty state after the first transition. Then w is recognized by $\mathcal{M}$. Conversely, given a trace $w \not\models \phi$, either $w = \epsilon$, or there exists $a \in \Sigma$ such that $a \preceq w$ and $a \nVdash \psi$. Neither will cause o_1 to be fired, so w is not recognized by $\mathcal{M}$. Therefore, C2.1 holds.

Let us prove that C2.2 also holds for $\mathcal{M}$. Suppose $\mathcal{M}$ detects no fulfillment of ψ at the first transition. That is, the first action a in a trace violates ψ (i.e., $a \Vdash\, !\, \psi$). Consequently, o_2 is fired, which removes o_1 and o_2, while adding back o_r. This traps $\mathcal{M}$ to a non-empty state. So C2.2 holds because $\mathcal{M}$ stops monitoring after one transition.

Case $[\psi]$. Suppose $\mathcal{M}$ is the OM returned from `compile`($[\psi]$) (refer to the corresponding compilation in Section 8.1).

First, let us prove that C1.1 holds for $\mathcal{M}$. Semantically, given a trace $w \models \phi$, either $w = \epsilon$, which is obviously accepted by $\mathcal{M}$, or there exists $a \in \Sigma$ such that $a \preceq w$ and $a \Vdash \psi$, which fires o_2 by the first transition. This removes both o_1 and o_2, and puts $\mathcal{M}$ to an empty state, so that any subsequent action will be accepted by $\mathcal{M}$. Then w is accepted by $\mathcal{M}$. Conversely, suppose $w \not\models \phi$. That is, there exists $a \in \Sigma$ such that $a \preceq w$ and $a \nVdash \psi$. Then the trace shall be rejected by $\mathcal{M}$ by the first transition. Therefore, C1.1 holds for $\mathcal{M}$.

C1.2 also holds for $\mathcal{M}$, because $\mathcal{M}$ enters an empty state after the first action been accepted.

Case $\neg\phi'$. Suppose $\mathcal{M}$ is the OM returned from `compile`($\neg\phi'$) and $\mathcal{M}'$ is the OM for ϕ' (refer to the corresponding compilation in Section 8.1).

1. If $\neg\phi'$ is typed as k-***mon***, ϕ' must be k-***enf***. Assume C1.1 and C1.2 holds for $\mathcal{M}'$. Let us prove that C2.1 and C2.2 hold for $\mathcal{M}$.

   By construction, the following correlations holds between $\mathcal{M}$ and $\mathcal{M}'$: given a finite trace $w \models \phi'$, for any $o \in \mathcal{O}'$, if $S'_0 \xrightarrow{w}_{\mathcal{M}'} S'_1$, then exists $S_1 \in \mathsf{states}_{\mathcal{M}}$, $S_0 \xrightarrow{w}_{\mathcal{M}} S_1$ satisfying:

   (a) $o \in S'_1 \Leftrightarrow o \in S_1 \backslash \{o^*\}$, and,

   (b) $o \in \mathsf{rule}_{\mathcal{M}'}(S'_1) \Rightarrow \iota'(o) = \iota(o)$, and,

   (c) $o \in \mathsf{cond}_{\mathcal{M}'}(S'_1).\, \iota'(o) = \psi \Rightarrow \iota(o) = \langle !\, \psi, \emptyset, \mathcal{O}' \uplus \{o^*_r\}\rangle$.

   Note that except for o^*, the obligation identifiers are identical in both OMs. This correlation implies that given the same trace, the violation of a simple condition in $\mathcal{M}'$ always coincides with the firing of a final trigger rule in $\mathcal{M}$. $\mathcal{M}$ enters an empty state iff some final trigger rule is fired (o^* can only be removed from the monitor state by the final trigger rules). However, a final trigger rule in $\mathcal{M}$ is fired iff there is an action a in a trace w satisfies $a \Vdash\, !\, \psi$ at some state S_1 of $\mathcal{M}$. With the correlation shown above, this action also violates the simple condition ψ at the corresponding state S'_1 of $\mathcal{M}'$ if w is inputed to $\mathcal{M}'$. Therefore, w is rejected by $\mathcal{M}'$, meaning that $w \models \neg\phi'$. That is, w is recognized by $\mathcal{M}$ iff $w \models \neg\phi'$. It is obvious that $\mathcal{M}$ is a pure OM. Therefore, C2.1 holds for $\mathcal{M}$.

   Let us prove C2.2 holds for $\mathcal{M}$. By the induction hyporeport, C1.2 holds for $\mathcal{M}'$, hence any state S'_1 of $\mathcal{M}'$ after k transitions contains no simple condition (otherwise a trace could also be rejected after k steps, which contradicts to the assumption that $\mathcal{L}_{acc}(\mathcal{M}')$ is k-bounded). Consequently, the corresponding state S_1 of $\mathcal{M}$ contains no final trigger rule (i.e., the final trigger rules are converted from the simple conditions in $\mathcal{M}'$, hence share the same identifiers). Therefore, w is never recognized after k transitions. C2.2 holds for $\mathcal{M}$.

2. Suppose $\neg\phi'$ is typed as k-***enf***, namely ϕ' is k-***mon***. By the induction hyporeport, C2.1 and C2.2 holds for $\mathcal{M}'$. Let us prove that C1.1 and C1.2 hold for $\mathcal{M}$.

   By construction, the following correlation between $\mathcal{M}$ and $\mathcal{M}'$ exists: given a finite trace $w \models \phi$, for any $o \in \mathcal{O}'$, if $S'_0 \xrightarrow{w}_{\mathcal{M}'} S'_1$, there exists $S \in \mathsf{states}_{\mathcal{M}}$, $S'_0 \xrightarrow{w}_{\mathcal{M}} S_1$ satisfying:

(a) $o \in S'_1 \Leftrightarrow o \in S_1$, and,

(b) $o \in \mathcal{O}'_f.\, \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \Rightarrow \iota(o) = \,!\, \psi$.

Note that although S' and S contains identical obligation identifiers, ι' and ι are indeed different. This correlation implies that by inputing the same trace to $\mathcal{M}'$ and $\mathcal{M}$, the firing of a final trigger rule in $\mathcal{M}'$ must coincide with the violation of a simple condition in $\mathcal{M}$.

A trace w is rejected by $\mathcal{M}$ iff there is a simple condition violated at some state S_1 of $\mathcal{M}$. According to the correlation shown above, a simple condition is violated at S_1 of $\mathcal{M}$ iff the corresponding final trigger rule is fired at S'_1 of $\mathcal{M}$. That is, w is recognized by $\mathcal{M}'$, meaning that $w \models \phi'$. On the whole, w is rejected by $\mathcal{M}$ iff $w \not\models \neg\phi'$. C1.1 holds for $\mathcal{M}$.

Let us prove that C1.2 also holds for $\mathcal{M}$. By the induction hyporeport, C2.2 holds for $\mathcal{M}'$, hence if $\mathcal{M}'$ does not detect the fulfillment of ϕ' within k steps, $\mathcal{M}'$ shall be trapped at an non-empty state without final trigger rules. According to the above correlation, there is no simple condition in any state of $\mathcal{M}$ after the k^{th} transition (i.e., because $\mathcal{M}'$ is a pure OM, each simple condition in $\mathcal{M}$ is converted from a final trigger rule in $\mathcal{M}'$). Therefore, C1.2 holds for $\mathcal{M}$.

Case $\phi' \wedge \phi''$. Suppose $\mathcal{M}$ is the OM returned from `compile`($\phi' \wedge \phi''$). Suppose further $\mathcal{M}'$ and $\mathcal{M}''$ are the OMs for ϕ' and ϕ''.

1. If ϕ' and ϕ'' are typed as k'-***enf*** and k''-***enf*** respectively, namely $\phi' \wedge \phi''$ is typed as ***max*** $(k',\ k'')$-***enf***, let us prove that C1.1 and C1.2 holds for $\mathcal{M}$.

   Obviously, C1.1 holds for $\mathcal{M}$ because the following statement holds for $\mathcal{M}$: given a finite trace $w \models \phi'$ and $w \models \phi''$, if $S'_0 \xrightarrow{w}_{\mathcal{M}'} S'_1$ and $S''_0 \xrightarrow{w}_{\mathcal{M}''} S''_1$, there exists $S_1 \in \mathsf{states}_{\mathcal{M}}$, $S_0 \xrightarrow{w}_{\mathcal{M}} S_1$ such that $S_1 = S'_1 \uplus S''_1$. That is, a trace w is rejected by $\mathcal{M}$ iff it violates a simple condition at some monitor state S_1 of $\mathcal{M}$. This means w violates a simple condition at either S'_1 or S''_1. In summary, a trace is rejected by $\mathcal{M}$ iff $w \not\models \phi' \wedge \phi''$.

   Let us prove that C1.2 also holds for $\mathcal{M}$. By the induction hyporeport, C1.2 holds for both $\mathcal{M}'$ and $\mathcal{M}''$. Therefore, for $\mathcal{M}'$, there is no simple condition in any state S'_1 after k' transitions; and for $\mathcal{M}''$, there is no simple conditions in any state S''_1 after k'' transitions. Therefore, after ***max*** $(k',\ k'')$ transitions, there will be no simple conditions in $S'_1 \uplus S''_1$. Thus, C1.2 holds for $\mathcal{M}$.

2. Suppose ϕ' and ϕ'' are typed as k'-***mon*** and k''-***mon***. Then $\phi' \wedge \phi''$ is typed as ***max*** $(k',\ k'')$-***mon***. Let us prove that C2.1 and C2.2 hold for $\mathcal{M}$.

   Firstly, $\mathcal{M}$ is a pure OM because both $\mathcal{M}'$ and $\mathcal{M}''$ are pure OMs, and identifying the set of final trigger rules for $\mathcal{M}$ will not induce simple conditions to the monitor states. By construction, $\mathcal{M}$ enters an empty state iff both $\mathcal{M}'$ and $\mathcal{M}''$ enter an empty state. Therefore, a trace is recognized by $\mathcal{M}$ iff it is recognized by both $\mathcal{M}'$ and $\mathcal{M}''$. Thus, C2.1 holds for $\mathcal{M}$.

   As mentioned above, $\mathcal{M}$ enters an empty state iff both $\mathcal{M}'$ and $\mathcal{M}''$ enter an empty state. By the induction hyporeport, C2.2 holds for both $\mathcal{M}'$ and $\mathcal{M}''$. If $\mathcal{M}$ fails to enter an empty state after ***max*** $(k',\ k'')$, then both $\mathcal{M}'$ and $\mathcal{M}''$ are trapped at a non-empty state, and thus $\mathcal{M}$ is trapped at an non-empty state.

Case $\phi' \vee \phi''$. This case is symmetrical to the case of $\phi' \wedge \phi''$.

Case $\mathsf{After}^+\ \phi' : \phi''$. Suppose $\mathcal{M}$ is the OM returned from `compile`($\mathsf{After}^+\ \phi' : \phi''$) (refer to the corresponding compilation in Section 8.1). And further suppose $\mathcal{M}'$ and $\mathcal{M}''$ are the OMs for ϕ' and ϕ'' respectively. Let us prove that C1.1 and C1.2 hold for $\mathcal{M}$.

By the compilation, $\mathcal{M}$ runs $\mathcal{M}'$ and $\mathcal{M}''$ in sequence, and $\mathcal{M}''$ is invoked iff some trigger rule $o \in \mathcal{O}'_f$ is fired. By the induction hyporeport, C2.1 holds for $\mathcal{M}'$, meaning that $\mathcal{M}'$ is a pure OM, so for any trace w, w will not be rejected during the monitoring of $\mathcal{M}'$. So a trace w is rejected by $\mathcal{M}$ iff it first fires a trigger rule $o \in \mathcal{O}'_f$ and later gets rejected by $\mathcal{M}''$. Because $o \in \mathcal{O}'_f$ can only be fired once, which terminate $\mathcal{M}'$ by erasing the state information of $\mathcal{M}$ and starts $\mathcal{M}''$ by putting $\mathcal{M}$ to the state S''_0 (the initial state of $\mathcal{M}''$), it must be fired by the *shortest* prefix u of w satisfying $u \models \phi'$ (i.e., $\forall u' \prec u.\, u' \not\models \phi'$). On the whole, a trace

w is rejected by $\mathcal{M}$ iff there exists $u \preceq w$ such that $u \models \phi'$ and for any $u' \prec u$, $u' \not\models \phi'$, and $w - u \not\models \phi''$, namely $w \not\models \mathsf{After}^+\ \phi' : \phi''$. So C1.1 holds for $\mathcal{M}$.

Let us prove that C1.2 also holds for $\mathcal{M}$. By the induction hyporeport, C2.2 holds for $\mathcal{M}'$ and C1.2 holds for $\mathcal{M}''$. If $\mathcal{M}$ detects no violation within $k' + k''$ steps, there are two possibilities: (1) $\mathcal{M}$ is trapped at a non-empty state and never starts $\mathcal{M}''$ (i.e., there is no $o \in \mathcal{O}'_f$ fired within k' steps). Of course, rejection of a trace will not occur after $k' + k''$ steps. (2) $\mathcal{M}''$ is started before k' steps, but there is no violation detected after $k' + k''$ steps. As a result, C1.2 holds for $\mathcal{M}$.

Case $\mathsf{After}^-\ \phi' : \phi''$. This case is basically the same as the one for After^+.

Case $\mathsf{Before}^+\ \phi_1 : \phi_2$. Suppose $\mathcal{M}$ is the OM returned from $\texttt{compile}(\mathsf{Before}^+\ \phi' : \phi'')$ (refer to the corresponding compilation in Section 8.1). Suppose further $\mathcal{M}'$ and $\mathcal{M}''$ are the OMs for ϕ' and ϕ'' respectively. Let us prove that C1.1 and C1.2 hold for $\mathcal{M}$.

By the compilation, $\mathcal{M}$ runs $\mathcal{M}'$ and $\mathcal{M}''$ in parallel. By the induction hyporeport, C2.1 holds for $\mathcal{M}'$, meaning that $\mathcal{M}'$ is a pure OM, so a trace is rejected iff it is rejected by $\mathcal{M}''$. If a trace is recognized by $\mathcal{M}'$, $o \in \mathcal{O}'_f$ is fired, which puts $\mathcal{M}$ to an empty state. Therefore, a trace is rejected by $\mathcal{M}$ iff it is rejected by $\mathcal{M}''$ before it is recognized by $\mathcal{M}'$. More formally, it means either there exists $u \preceq w$ such that $u \models \phi'$ and for all $u' \prec u$, $u' \not\models \phi'$, and $u \not\models \phi''$, or there is no such $u \preceq w$, $u \models \phi'$, and $w \not\models \phi''$. That is, $w \not\models \mathsf{Before}^+\ \phi' : \phi''$. So C1.1 holds for $\mathcal{M}$.

Let us prove that C1.2 holds for $\mathcal{M}$. Suppose $\mathcal{M}$ detects no violation within k'' steps. By the induction hyporeport, $\mathcal{M}''$ shall stop monitoring in k'' steps. As mentioned above, a trace is rejected by $\mathcal{M}$ iff it is rejected by $\mathcal{M}''$. So C1.2 holds for $\mathcal{M}$.

Case $\mathsf{Before}^-\ \phi' : \phi''$. This case is similar to the case for Before^+.

Case $\mathsf{Fulfilling}^{k'}\ \phi' ?\, \phi''\ :\ \phi'''$. Suppose $\texttt{compile}(\mathsf{Fulfilling}^{k'}\ \phi' ?\, \phi'' : \phi''')$ gives us $\mathcal{M}$. Suppose further $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$, $\texttt{compile}(\phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}'', \emptyset \rangle$ and $\texttt{compile}(\phi''') = \langle \boldsymbol{enf}, k''', \mathcal{M}''', \emptyset \rangle$. Let us prove that C1.1 and C1.2 hold of $\mathcal{M}$.

$\mathcal{M}$ runs $\mathcal{M}'$ and another OM which is either $\mathcal{M}''$ or $\mathcal{M}'''$ in sequence. By the induction hyporeport, C2.1 holds for $\mathcal{M}'$, meaning that $\mathcal{M}'$ is a pure OM. So rejection of a trace can never occur during the monitoring of $\mathcal{M}'$. Therefore, a trace is reject by $\mathcal{M}$ iff it is either rejected by $\mathcal{M}''$ or $\mathcal{M}'''$. By construction, $\mathcal{M}$ enters either S''_0, (the initial state of $\mathcal{M}''$) or S'''_0, (the initial state of $\mathcal{M}'''$). If a trace w is reject by $\mathcal{M}$ because it is rejected by $\mathcal{M}''$, there exists a shortest prefix u of w, which is recognized by $\mathcal{M}'$, and $w - u$ is rejected by $\mathcal{M}''$. Otherwise, a trace w is reject by $\mathcal{M}$ because it is rejected by $\mathcal{M}'''$, there exists a prefix u such that $|u| = k$ and u is not recognized by $\mathcal{M}'$, and $w - u$ is rejected by $\mathcal{M}'''$. This reflecting to the semantics indicates that $w \not\models \mathsf{Fulfilling}^{k'}\ \phi' ?\, \phi'' : \phi'''$. Thus, C1.1 holds for $\mathcal{M}$.

Let us prove that C1.2 holds for $\mathcal{M}$. Suppose $\mathcal{M}$ detects no violation within $k' + \boldsymbol{max}\,(k'',\ k''')$. First, after k' steps, $\mathcal{M}''$ or $\mathcal{M}'''$ must have been started by $\mathcal{M}$. By the induction hyporeport, C2.1 holds for both $\mathcal{M}''$ and $\mathcal{M}'''$. Thus, after $\boldsymbol{max}\,(k',\ k'')$, if no violation has been detected, both $\mathcal{M}''$ and $\mathcal{M}'''$ will stop enforcing. Therefore, C1.2 holds for $\mathcal{M}$.

Case $\mathsf{Eventually}\ \phi'$. Suppose $\mathcal{M}$ is the OM obtained from $\texttt{compile}(\mathsf{Eventually}\ \phi')$. Suppose further $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f \rangle$. Then let us prove C2.1 holds for $\mathcal{M}$.

By the induction hyporeport, C2.1 holds for $\mathcal{M}'$, meaning that $\mathcal{M}'$ is a pure OM. By the compilation algorithm, $\mathcal{M}$ is to start a new version of $\mathcal{M}'$ at each state, so $\mathcal{M}$ is also a pure OM. $\mathcal{M}$ enters an empty state iff a final trigger rule $o \in \uplus\{\mathcal{O}'^{i}_f | i = 1, 2, \cdots, k'\}$ is fired, meaning that there must be a suffix that is recognized by a version of $\mathcal{M}'$. That is, $w \models \mathsf{Eventually}\ \phi'$. C2.1 holds for $\mathcal{M}$.

Cases $\mathsf{Always}\ \phi'$. The case is similar to the one for $\mathsf{Eventually}$.

Case Ignoring $\psi : \phi'$. The proof for this case is rather straightforward. Suppose $\mathcal{M}'$ is the OM obtained from `compile`(ϕ'). By the compilation algorithm, the actions that satisfies ψ has no effect to $\mathcal{M}'$ (i.e., does not cause violation, and does not fire trigger rules). By the induction hyporeport, C1.1 or C2.1 holds for $\mathcal{M}'$ depending on the typing of ϕ', C1.1 or C2.1 shall also hold for $\mathcal{M}$.