Enforcement Monitoring wrt. the Safety-Progress Classification of Properties

Yliès Falcone
VERIMAG, U of Grenoble I
Grenoble, France
(33) 4 56 52 04 13
Ylies.Falcone@imag.fr

Jean-Claude Fernandez
VERIMAG, U of Grenoble I
Grenoble, France
(33) 4 56 52 03 79
Jean-Claude.Fernandez@imag.fr

Laurent Mounier
VERIMAG, U of Grenoble I
Grenoble, France
(33) 4 56 52 03 54
Laurent.Mounier@imag.fr

ABSTRACT

Runtime enforcement is a powerful technique to ensure that a program will respect a given set of properties. We extend previous works on this topic in several directions. Firstly, we propose a generic notion of enforcement monitors based on a memory device and finite sets of control states and enforcement operations. Moreover, we specify their enforcement abilities wrt. the *general* safety-progress classification of properties. Furthermore, we propose a *systematic* technique to produce an enforcing monitor from the automaton recognizing a given safety, guarantee, or response property. Finally, we depict a prototype toolbox implementing the features proposed in this paper.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, validation, reliability*

General Terms

Verification, Theory

Keywords

runtime enforcement, monitor, property, safety-progress, synthesis

1. INTRODUCTION

The growing complexity of nowadays programs and systems induces a rise of needs in validation. With the enhancement of engineering methods, software components tend to be more and more reusable. When retrieving an external component, the question of how this code meets a set of proper requirements raises. Using formal methods appears as a solution to provide techniques to regain the needed confidence. However, these techniques should remain practical enough to be adopted by software engineers.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

Runtime monitoring falls it this category. It consists in supervising at runtime the execution of an underlying program against a set of expected properties. With an appointed monitor, one is able to detect any occurrence of specific property violations. Such a detection might be a sufficient assurance. However, for certain kind of systems a misbehavior might be not acceptable. To prevent this, a possible solution is then to *enforce* the desired property: the monitor not only observes the current program execution, but it also controls it in order to ensure that the expected property is fulfilled. Such a control should usually remain *transparent*, meaning that it should always output the *longest correct prefix* of the original execution sequences.

Runtime enforcement monitoring was initiated by the work of Schneider [17] on what has been called *security automata*. In this work the monitors watch the current execution sequence and halt the underlying program whenever it deviates from the desired property. Such security automata are able to enforce the class of safety properties [10], stating that *something bad can never happen*. Later, Viswanathan [20] noticed that the class of enforceable properties is impacted by the computational power of the enforcement monitor. As the enforcement mechanism can implement no more than computable functions, the enforceable properties are included in the decidable ones. More recently [14, 15], Ligatti and al. showed that it is possible to enforce at runtime more than safety properties. Using a more powerful enforcement mechanism called *edit-automata*, it is possible to enforce the larger class of *infinite renewal properties*, able to express some kinds of *obligations* used in security policies. More than simply halting an underlying program, edit-automata can also "suppress" (i.e., freeze) and "insert" (frozen) actions in the current execution sequence. To better cope with practical resource constraints, Fong [9] studied the effect of memory limitations on enforcement mechanisms. He introduced the notion of *Shallow History Automata* which are only aware of the occurrence of past events, and do not keep any information about the order of their arrival. He showed that such a "shallow history" indeed leads to some computational limitations for the enforced properties. However, many interesting properties remain enforceable using shallow history automata. The runtime enforcement monitoring approach was implemented in numerous tools, *e.g.* [7, 5]. Most of them are based more or less on security automata, whereas [13] introduces a more expressive framework based on edit-automata.

In this paper, we propose to extend these previous works in several directions. Firstly, we study the enforcement ca-

pabilities relatively to the so-called *safety-progress* hierarchy of properties [3, 4]. This classification differs from the more classical safety-liveness classification [12, 1] by offering a rather clear characterization of a number of interesting kinds of properties (*e.g.* obligation, accessibility, justice, etc.). Thus, it provides a finer-grain classification of enforceable properties. Moreover, in this safety-progress hierarchy, each property φ can be characterized by a particular kind of (finite state) recognizing automaton $\mathcal{A}_\varphi$. Secondly, we show how to generate an enforcement monitor for φ in a *systematic way*, from a recognizing automaton $\mathcal{A}_\varphi$. This enforcement monitor is based on a *finite set of control states*, and an *auxiliary memory*. This general notion of enforcing monitor encompasses the previous notions of security automata, edit-automata and "shallow history" automata. Moreover, we have implemented a prototype toolbox to support the concepts introduced in this paper (in particular the generation of enforcement monitors from recognizing automata). The underlying technology we chose to implement the enforcement mechanism is based on Aspect Oriented Programming. Indeed, inlining the enforcement monitor seems to be one of the most promising approaches for enforcement monitoring [6].

The remainder of this article is organized as follows. The Sect. 2 introduces some preliminary notions for our work. In Sect. 3 we recall briefly the necessary elements from the safety-progress classification of properties. Then, we present our notion of enforcement monitor and their properties in Sect. 4. The Sect. 5 studies the enforcement capability wrt. the classes of the safety-progress classification and Sect. 6 compares these results with similar works. In Sect. 7 we give some insights about the prototype tool we developed to experiment this approach. Finally, the Sect. 8 exposes some concluding remarks.

A companion report [8] provides more details and complete proofs of the theorems introduced in this paper.

2. PRELIMINARIES AND NOTATIONS

This section introduces some preliminary notations, namely the notions of *program execution sequences* and *program properties* we will consider in the remainder of this article.

2.1 Sequences, and execution sequences

Considering a finite set of elements E, we define notations about sequences of elements belonging to E. A sequence σ containing elements of E is formally defined by a function $\sigma : \mathbb{N} \to E$ where $\mathbb{N}$ is the set of natural numbers. We denote by E^* the set of finite sequences over E (partial function from $\mathbb{N}$), and by E^ω the set of infinite sequences over E (total function from $\mathbb{N}$). The set $E^\infty = E^* \cup E^\omega$ is the set of all sequences (finite or not) over E. The empty sequence is denoted ϵ. The length (number of elements) of a finite sequence σ is noted $|\sigma|$ and the $(i+1)$-th element of σ is denoted by σ_i. For two sequences $\sigma \in E^*, \sigma' \in E^\infty$, we denote by $\sigma \cdot \sigma'$ the concatenation of σ and σ', and by $\sigma \prec \sigma'$ (resp. $\sigma' \succ \sigma$) the fact that σ is a strict prefix of σ' (resp. σ' is a strict suffix of σ). The sequence σ is said to be a strict prefix of $\sigma' \in \Sigma^\infty$ when $\forall i \in \{0, \ldots, |\sigma| - 1\} \cdot \sigma_i = \sigma'_i$. When $\sigma' \in E^*$, we note $\sigma \preceq \sigma' \stackrel{\text{def}}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$.

A program $\mathcal{P}$ is considered as a generator of execution sequences. We are interested in a restricted set of operations the program can perform. These operations influence the truth value of properties the program is supposed to fulfill.

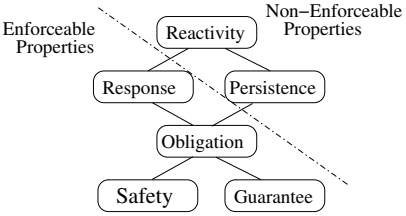

Figure 1: The safety-progress classification of properties

We abstract these operations by a finite set of *events*, namely a vocabulary Σ. We denote by $\mathcal{P}_\Sigma$ a program for which the vocabulary is Σ. The set of execution sequences of $\mathcal{P}_\Sigma$ is denoted $Exec(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$.

2.2 Properties as sets of execution sequences

A property φ is defined as a set of execution sequences, *i.e.* $\varphi \subseteq \Sigma^\infty$. Considering a given execution sequence σ, when $\sigma \in \varphi$ (noted $\varphi(\sigma)$), we say that σ *satisfies* φ. A consequence of this definition, noticed in [14, 17], is that properties we will consider are restricted to *single* execution sequences, excluding specific properties defined on powersets of execution sequences (like fairness, for instance). Finally, for a property φ and an execution sequence σ we denote by $Pref(\varphi, \sigma)$ the set of prefixes of σ that belong to φ, and by $Max_\preceq(Pref(\varphi, \sigma))$ its longest element.

As noticed in [14], a property can be effectively enforced at runtime only if it is *reasonable* in the following sense: deciding if any prefix of an execution sequence satisfies this property should be a computable function.

3. A SAFETY-PROGRESS CLASSIFICATION OF PROPERTIES

This section recalls and extends some results about the safety-progress [3, 4] classification of properties. In the original papers this classification introduced a hierarchy between properties defined as *infinite* execution sequences. We extend the classification to deal with finite-length execution sequences.

3.1 Generalities about the classification

The safety-progress classification (SP classification, for short) is constituted of four basic classes defined over infinite execution sequences. Informally:

- *safety* properties are the properties for which whenever a sequence satisfies a property, *all its prefixes* satisfy this property.
- *guarantee* properties are the properties for which whenever a sequence satisfies a property, *there are some prefixes* (at least one) satisfying this property.
- *response* properties are the properties for which whenever a sequence satisfies a property, *an infinite number of its prefixes* satisfy this property.
- *persistence* properties are the properties for which whenever a sequence satisfies a property, *all its prefixes* continuously satisfy this property from a certain point.

Furthermore, two extra classes can be defined as (finite) boolean combinations of basic classes. The *Obligation class* can be defined as the class obtained by positive boolean combination of safety and guarantee properties. The *Reactivity class* can be defined as the class obtained by positive boolean combination of response and persistence properties.

Given a set of events Σ, we note Safety_Σ (resp. Guarantee_Σ, Response_Σ, $\text{Persistence}_\Sigma$) the set of safety (resp. guarantee, response, persistence) properties defined over Σ.

The safety-progress classification is an alternative to the more classical safety-liveness [12, 1] dichotomy. Unlike the latter, the safety-progress classification is a hierarchy and not a partition. It provides a finer-grain classification, and the properties of each class are characterized according to four *views* [3]. We will consider here only the automata view.

3.2 The automata view

For each class of the safety-progress classification it is possible to syntactically characterize a recognizing automaton. We define here a variant of deterministic and complete Streett automata ([18, 3]).

DEFINITION 3.1 (STREETT AUTOMATON). *A Streett automaton is a tuple* $(Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_1, P_1), \ldots, (R_n, P_n)\})$ *defined relatively to a set of events* Σ. *The finite set* Q *is the set of automaton states, where* $q_{\text{init}} \in Q$ *is the initial state. The function* $\longrightarrow : Q \times \Sigma \to Q$ *is the transition function. In the following, for* $q, q' \in Q, e \in \Sigma$ *we abbreviate* $\longrightarrow (q, e) = q'$ *by* $q \xrightarrow{e} q'$. *The set* $\{(R_1, P_1), \ldots, (R_m, P_m)\}$ *is the set of accepting pairs, in which for all* $i \leq m$, $R_i \subseteq Q$ *are the sets of recurrent states, and* $P_i \subseteq Q$ *are the sets of persistent states.*

We refer an automaton with m accepting pairs as a m-automaton. When $m = 1$, a 1-automaton is also called a *plain*-automaton, and we refer R_1 and P_1 as R and P. In the following (otherwise mentioned) $\sigma \in \Sigma^\omega$ designates an execution sequence of a program, and $\mathcal{A} = (Q^{\mathcal{A}}, {q_{\text{init}}}^{\mathcal{A}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ a Streett m-automaton.

The *run* of σ on $\mathcal{A}$ is the sequence of states involved by the execution of $\mathcal{A}$ when σ is inputted. It is formally defined as $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$ where $\forall i \cdot (q_i \in Q^{\mathcal{A}} \wedge q_i \xrightarrow{\sigma_i}_{\mathcal{A}} q_{i+1}) \wedge q_0 = {q_{\text{init}}}^{\mathcal{A}}$. Also, we consider the notion of infinite visitation of an execution sequence σ on a Streett automaton $\mathcal{A}$, denoted $inf(\sigma, \mathcal{A})$, as the set of states appearing infinitely often in $run(\sigma, \mathcal{A})$.

For a Streett automaton, independently of the class of recognized property, the notion of acceptance condition is defined using the accepting pairs[1].

DEFINITION 3.2 (ACCEPTANCE CONDITION (OVER Σ^ω)). *We say that* $\mathcal{A}$ *accepts* $\sigma \in \Sigma^\omega$ *if* $\forall i \in \{1, \ldots, m\} \cdot inf(\sigma, \mathcal{A}) \cap R_i \neq \emptyset \vee inf(\sigma, \mathcal{A}) \subseteq P_i$.

By setting syntactic restrictions on a Streett automaton, we characterize the class of properties recognized by such an automaton.

- A *safety automaton* is a plain automaton such that $R = \emptyset$ and there is no transition from a state $q \in \overline{P}$ to a state $q' \in P$.
- A *guarantee automaton* is a plain automaton such that $P = \emptyset$ and there is no transition from a state $q \in R$ to a state $q' \in \overline{R}$.
- A *response automaton* is a plain automaton such that $P = \emptyset$.
- A *persistence automaton* is a plain automaton such that $R = \emptyset$.

We say that a Streett automaton $\mathcal{A}_\varphi$ *defines* a property φ if and only if the set of execution sequences accepted by $\mathcal{A}_\varphi$ is equal to φ. Conversely, a property $\varphi \subseteq \Sigma^\infty$ is said to be *specifiable* by an automaton if the set of execution sequences accepted by the automaton is φ.

We extend the capability of Streett automata in order to deal also with finite-length execution sequences. As informally stated in the introduction, when enforcing a property on a program exhibiting an incorrect behavior, we want the enforcement monitor to release a prefix (the longest) satisfying the property. Hence, we need a notion of acceptance for finite execution sequences.

DEFINITION 3.3 (ACCEPTANCE CONDITION (OVER Σ^*)). *For a finite-length execution sequence* $\sigma \in \Sigma^*$ *such that* $|\sigma| = n$, *we say that the* m*-automaton* $\mathcal{A}$ *accepts* σ *if* $(\exists q_0, \ldots, q_{n-1} \in Q^{\mathcal{A}} \cdot run(\sigma, \mathcal{A}) = q_0 \cdots q_{n-1} \wedge q_0 = {q_{\text{init}}}^{\mathcal{A}}$ *and* $\forall i \in \{1, \ldots, m\} \cdot q_{n-1} \in P_i \cup R_i)$.

This acceptance condition matches the definition of finitary properties defined in [3]. We can now define the safety-progress classification for both finite and infinite sequences.

In the remainder of this article, we will only consider plain Streett automata. Additional details for the following developments for general Streett automata can be found in [8]. As a consequence, we will only deal with the classes of safety, guarantee, response and persistence properties.

DEFINITION 3.4. *A property* φ *that is specifiable by an automaton is a* κ*-property if the automaton is a* κ*-automaton, where* $\kappa \in \{\text{safety}, \text{guarantee}, \text{response}, \text{persistence}\}$

A graphical representation of the safety-progress classification of properties is depicted on Fig. 1. This illustrates the hierarchal organization of this classification, *e.g.* safety properties are response and persistence properties, etc.

4. ENFORCEMENT MONITORS

A program $\mathcal{P}$ is considered as a generator of execution sequences. We want to build an enforcement monitor (EM) for a property φ such that the two following constraints hold:

soundness: any execution sequence allowed by the EM should satisfy φ;

transparency: execution sequences should be modified in a minimal way, namely if a sequence already satisfies φ it should remain unchanged, otherwise its *longest prefix* satisfying φ should be allowed by the EM.

4.1 Enforcement monitors

We define now the central notion of enforcement monitor. Such a runtime device monitors a target program by watching its relevant events. On each inputted event its current state evolves and an enforcement operation is performed.

[1]There are several equivalent acceptance conditions for Streett automata. Here we follow [3].

$$\sigma \Downarrow_{\mathcal{A}_\downarrow} o \quad (1)$$

$$\varphi(\sigma) \Rightarrow \sigma = o \quad (2)$$

$$\neg\varphi(\sigma) \wedge Pref(\varphi,\sigma) \neq \emptyset \Longrightarrow o = Max_{\preceq}(Pref(\varphi,\sigma)) \quad (3)$$

$$\neg\varphi(\sigma) \wedge Pref(\varphi,\sigma) = \emptyset \Longrightarrow o = \epsilon \quad (4)$$

Figure 2: Constraints for enforcement on finite sequences

DEFINITION 4.1 (ENFORCEMENT MONITOR (EM)). *An* EM $\mathcal{A}_\downarrow$ *is a 4-tuple* $(Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, Stop^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$ *defined relatively to a set of events* Σ *and a set of enforcement operations Ops. The finite set* $Q^{\mathcal{A}_\downarrow}$ *denotes the control states,* ${q_{\text{init}}}^{\mathcal{A}_\downarrow} \in Q^{\mathcal{A}_\downarrow}$ *is the initial state and* $Stop^{\mathcal{A}_\downarrow}$ *is the set of stopping states* ($Stop^{\mathcal{A}_\downarrow} \subseteq Q^{\mathcal{A}_\downarrow}$). *The function* $\longrightarrow_{\mathcal{A}_\downarrow}$: $Q^{\mathcal{A}_\downarrow} \times \Sigma \to Q^{\mathcal{A}_\downarrow} \times Ops$ *is the transition function. In the following we abbreviate* $\longrightarrow_{\mathcal{A}_\downarrow}(q,a) = (q',\alpha)$ *by* $q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q'$. *We also assume that outgoing transitions from a stopping state only lead to another stopping state:* $\forall q \in Stop^{\mathcal{A}_\downarrow} \cdot \forall a \in \Sigma \cdot \forall \alpha \in Ops \cdot \forall q' \in Q^{\mathcal{A}_\downarrow} \cdot q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q' \Rightarrow q' \in Stop^{\mathcal{A}_\downarrow}$.

Typical enforcement operations allow the EM either to halt the target program (when the current input sequence irreparably violates the property), or to store the current event in a *memory device* (when a decision has to be postponed), or to dump the content of the memory device (when the target program went back to a safe behavior). We first give a more precise definition of such enforcement operations, then we formalize the way an EM reacts to an input sequence provided by a target program through the standard notions of *configuration* and *derivation*.

DEFINITION 4.2 (ENFORCEMENT OPERATIONS). *Each enforcement operation takes as inputs an event and a memory content (i.e., a sequence of events) to produce a new memory content and an output sequence:* $Ops \subseteq 2^{(\Sigma\times\Sigma^*)\to(\Sigma^*\times\Sigma^*)}$. *In the following we consider a set* $Ops = \{halt, store, dump\}$ *defined as follows:*

- $halt(a,m) = (m,\varepsilon)$,
- $store(a,m) = (m.a,\varepsilon)$,
- $dump(a,m) = (\varepsilon, m.a)$.

DEFINITION 4.3 (EM CONFIGURATIONS, DERIVATIONS). *For an EM* $\mathcal{A}_\downarrow = (Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, Stop^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$, *a* configuration *is a triplet* $(q,\sigma,m) \in Q^{\mathcal{A}_\downarrow} \times \Sigma^* \times \Sigma^*$ *where* q *denotes the current control state,* σ *the current input sequence, and* m *the current memory content.*
We say that a configuration (q',σ',m') *is* derivable in one step *from the configuration* (q,σ,m) *and* produces the output $o \in \Sigma^*$, *and we note* $(q,\sigma,m) \overset{o}{\hookrightarrow} (q',\sigma',m')$ *if and only if* $\sigma = a.\sigma' \wedge q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q' \wedge \alpha(a,m) = (m',o)$.
We say that a configuration C' *is* derivable in several steps *from a configuration* C *and* produces the output $o \in \Sigma^*$, *and we note* $C \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} C'$, *if and only if there exits* $k \geq 0$ *and configurations* $C_0, C_1, \ldots, C_k$ *such that* $C = C_0$, $C' = C_k$, $C_i \overset{o_i}{\hookrightarrow} C_{i+1}$ *for all* $0 \leq i < k$, *and* $o = o_0 \cdot o_1 \cdots o_{k-1}$.
Also the configuration C *is derivable from itself in one step and produces the output* ϵ, *we note* $C \overset{\epsilon}{\Rightarrow} C$.

4.2 Enforcing a property

We now describe how an EM can enforce a property on a given program. The notion of enforcement is based on how a monitor transforms a given inputted sequence into an output sequence. For the upcoming definitions we will distinguish between finite and infinite sequences. In the following, we consider an EM $\mathcal{A}_\downarrow = (Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, Stop^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$.

DEFINITION 4.4 (SEQUENCE TRANSFORMATION). *We say that:*
The sequence $\sigma \in \Sigma^*$ *produced by* $\mathcal{P}_\Sigma$ *is transformed by* $\mathcal{A}_\downarrow$ *into the sequence* $o \in \Sigma^*$, *which is noted* $({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma) \Downarrow_{\mathcal{A}_\downarrow} o$, *if* $\exists q \in Q^{\mathcal{A}_\downarrow}, m \in \Sigma^*$ *such that* $({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma, \epsilon) \overset{o}{\Longrightarrow}_{\mathcal{A}_\downarrow} (q,\epsilon,m)$.
The sequence $\sigma \in \Sigma^\omega$ *is transformed by* $\mathcal{A}_\downarrow$ *into the sequence* $o \in \Sigma^\infty$, *which is noted* $({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma) \Downarrow_{\mathcal{A}_\downarrow} o$, *if* $\forall \sigma' \in \Sigma^* \cdot \sigma' \prec \sigma \cdot (\exists o' \in \Sigma^* \cdot ({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma') \Downarrow_{\mathcal{A}_\downarrow} o' \wedge o' \preceq o)$.

We define the notion of property-enforcement by an EM.

DEFINITION 4.5 (PROPERTY-ENFORCEMENT). *Let us consider a property* φ, *we say that* $\mathcal{A}_\downarrow$ *enforces the property* φ *on a program* $\mathcal{P}_\Sigma$ *(noted* $Enf(\mathcal{A}_\downarrow, \varphi, \mathcal{P}_\Sigma)$*) iff*

- $\forall \sigma \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^*, \exists o \in \Sigma^* \cdot enforced(\sigma, o, \mathcal{A}_{\downarrow\varphi}, \varphi)$, *where the predicate* $enforced(\sigma, o, \mathcal{A}_{\downarrow\varphi}, \varphi)$ *corresponds to the conjunction of the constraints of Fig. 2.*
- $\forall \sigma' \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega, \forall \sigma \prec \sigma' \cdot enforced(\sigma, o, \mathcal{A}_{\downarrow\varphi}, \varphi)$.

(1) *ensures soundness, while* (2), (3), (4) *ensure transparency of* $\mathcal{A}_\downarrow$. *Indeed:* (1) *stipulates that the sequence* σ *is transformed by* $\mathcal{A}_\downarrow$ *into a sequence* o; (2) *ensures that if* σ *satisfied already the property then it is not transformed; and in the case where* σ *does not satisfy* φ, *if there exists a prefix of* σ *satisfying the property* (3) *ensures that* o *is the longest prefix of* σ *satisfying the property, else* (4) *ensures that* $\mathcal{A}_\downarrow$ *produces* ϵ.

EXAMPLE 4.1 (ENFORCEMENT MONITOR). *Considering a set of events* $\Sigma = \{a,b,c\}$,

- *at the bottom of Fig. 3 is depicted an EM enforcing the property expressed by the* ω*-regular expression* $b^* \cdot a^+ \cdot b \cdot \Sigma^\omega$;
- *and Fig. 4 (right-hand side) shows an EM enforcing the property* $(a^* \cdot b)^\omega$.

5. ENFORCEMENT WRT. THE SP CLASSIFICATION

We now study how to practically enforce properties of the safety-progress hierarchy (Sect. 3). More precisely, we show which classes of properties can be effectively enforced by an

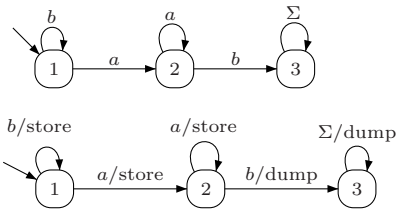

Figure 3: A guarantee-automaton and the corresponding EM for $b^* \cdot a^+ \cdot b \cdot \Sigma^\omega$

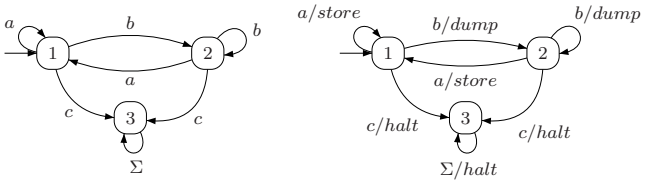

Figure 4: A response-automaton and the corresponding EM for $(a^* \cdot b)^\omega$

EM, and more important, we provide a systematic construction of an EM for a property φ from the Streett automaton defining this property. This construction technique is specific to each class of properties. We consider below a Streett plain-automaton $\mathcal{A}_\varphi = (Q^{\mathcal{A}_\varphi}, {q_{\text{init}}}^{\mathcal{A}_\varphi}, \Sigma, \longrightarrow_{\mathcal{A}_\varphi}, (R, P))$.

5.1 Enforcement monitor synthesis

We define three general operations whose purpose is to transform a Streett plain-automaton recognizing a safety (resp. guarantee, response) property into an enforcement monitor enforcing the same property.

Definition 5.1 (Safety Transformation). *Let $\mathcal{A}_\varphi$ a safety-automaton, and φ a reasonable safety property, we define a transformation* TSaf *of this automaton into an EM $\mathcal{A}_{\downarrow\varphi} = (Q^{\mathcal{A}_{\downarrow\varphi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}}, Stop^{\mathcal{A}_{\downarrow\varphi}}, \longrightarrow_{\mathcal{A}_{\downarrow\varphi}})$ such that $Q^{\mathcal{A}_{\downarrow\varphi}} = Q^{\mathcal{A}_\varphi}$, ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}} = {q_{\text{init}}}^{\mathcal{A}_\varphi}$ (with ${q_{\text{init}}}^{\mathcal{A}_\varphi} \in P$), $Stop^{\mathcal{A}_{\downarrow\varphi}} = \overline{P}$. The transition relation $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ is defined from $\longrightarrow_{\mathcal{A}_\varphi}$ as the smallest relation verifying the following rules:*

- $q \xrightarrow{a/dump}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \in P \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q'$
- $q \xrightarrow{a/halt}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin P \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q'$

We note $\mathcal{A}_{\downarrow\varphi} = \text{TSaf}(\mathcal{A}_\varphi)$.

An EM obtained via the TSaf transformation processes the inputted execution sequence and enforces the expected property. Informally it can be understood as follows. While the current execution sequence satisfies the underlying property, it dumps directly the input on its output. Once the execution sequence deviates from the property, it halts immediately the underlying program with a *halt* operation.

We now define a similar transformation for *guarantee* properties. The TGuar transformation uses the set $Reach_{\mathcal{A}_\varphi}(q)$ of reachable states from a state q. Given a Streett automaton $\mathcal{A}_\varphi$ with a set of states $Q^{\mathcal{A}_\varphi}$, we have $\forall q \in Q^{\mathcal{A}_\varphi} \cdot Reach_{\mathcal{A}_\varphi}(q) = \{q' \in Q^{\mathcal{A}_\varphi} \mid \exists (q_i)_i, (a_i)_i, \cdot q \xrightarrow{a_0}_{\mathcal{A}_\varphi} q_0 \xrightarrow{a_1}_{\mathcal{A}_\varphi} q_1 \cdots q'\}$.

Definition 5.2 (Guarantee Transformation). *Let $\mathcal{A}_\varphi$ a guarantee-automaton recognizing a property $\varphi \in \text{Guarantee}_\Sigma$. We define a transformation* TGuar *of this automaton into an EM $\mathcal{A}_{\downarrow\varphi} = (Q^{\mathcal{A}_{\downarrow\varphi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}}, Stop^{\mathcal{A}_{\downarrow\varphi}}, \longrightarrow_{\mathcal{A}_{\downarrow\varphi}})$ such that $Q^{\mathcal{A}_{\downarrow\varphi}} = Q^{\mathcal{A}_\varphi}$, ${q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}} = {q_{\text{init}}}^{\mathcal{A}_\varphi}$, and $Stop^{\mathcal{A}_{\downarrow\varphi}} = \{q \in Q^{\mathcal{A}_{\downarrow\varphi}} \mid \nexists q' \in Reach_{\mathcal{A}_\varphi}(q) \wedge q' \in R\}$. The transition relation $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ is defined from $\longrightarrow_{\mathcal{A}_\varphi}$ as the smallest relation verifying the following rules:*

- $q \xrightarrow{a/dump}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \in R \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q'$
- $q \xrightarrow{a/halt}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q' \wedge \nexists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$
- $q \xrightarrow{a/store}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q' \wedge \exists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$

Note that there is no transition from $q \in R$ to $q' \in \overline{R}$. And, as $P = \emptyset$, we do not have transition from $q \in P$ to $q' \in P$. We note $\mathcal{A}_{\downarrow\varphi} = \text{TGuar}(\mathcal{A}_\varphi)$.

Informally this can be understood as follows. While the current execution sequence does not satisfy the underlying property, it stores each event of the input sequence. Once, the execution sequence satisfies the property, it dumps the content of the memory and the events stored so far. Note that an automaton resulting of this transformation satisfies the enforcement monitor constraints (notably the stopping states constraint). The following example illustrates this principle.

Example 5.1 (Guarantee Transformation). *Considering a set of events $\Sigma = \{a, b\}$, Fig. 3 (up-side) shows a Streett automaton recognizing the guarantee property expressed by $b^* \cdot a^+ \cdot b \cdot \Sigma^\omega$. Its set of states is $\{1, 2, 3\}$, the initial state is 1, and we have $R = \{3\}$ and $P = \emptyset$. Below is depicted the EM enforcing the same property, obtained by the* TGuar *transformation. One can notice there is no stopping state.*

Finding a transformation for a *response* property φ needs to slightly extend the definition of TGuar to deal with transitions of a Streett automaton leading from states belonging to R to states belonging to $\overline{R}$ (since such transitions are absent when φ is a *guarantee* property). Therefore, we introduce a new transformation called TResp obtained from the TGuar transformation (Def. 5.2) by adding a rule to deal with the aforementioned difference.

Definition 5.3 (Response Transformation). *Let $\mathcal{A}_\varphi$ a response-automaton recognizing a property $\varphi \in \text{Response}_\Sigma$. We define a transformation* TResp *of this automaton into an enforcement monitor $\mathcal{A}_{\downarrow\varphi}$ by adding the following rules to the* TGuar *transformation to define $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ from $\rightarrow_{\mathcal{A}_\varphi}$:*

- $q \xrightarrow{a/store}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q' \wedge \exists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$
- $q \xrightarrow{a/halt}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \xrightarrow{a}_{\mathcal{A}_\varphi} q' \wedge \nexists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$

An EM obtained via the TResp transformation processes the inputted execution sequence and enforces the originally recognized property. Informally the principle is similar to the one of guarantee enforcement, except that there might be an alternation in the run between states of R and $\overline{R}$. While the current execution sequence does not satisfy the underlying property (the current state is in $\overline{R}$), it stores each event of the input sequence. Once, the execution sequence satisfies the property (the current state is in R), it dumps the content of the memory and the events stored so far. On Fig. 4 we illustrate the TResp transformation of a Streett response-automaton (left-hand side) recognizing the property $(a^* \cdot b)^\omega$ into an EM enforcing the same property.

Obligation properties. The more complex transformation of obligation-automata into EMs can be found in the companion report [8] of this paper. Roughly speaking, this transformation is based on the accepting pairs of Streett automaton. It consists in applying the TSaf and TGuar transformations respectively on the persistent and recurrent states.

5.2 Enforcement wrt. the SP classification

Using the aforementioned transformations it is possible to derive an EM of a certain property from a recognizing automaton for this (enforceable) property. In the following, we characterize the set of enforceable properties wrt. the safety-progress classification. Fig. 1 delineates the enforceable classes of the safety-progress classification.

5.2.1 Enforceable properties.

The safety (resp. guarantee, obligation and response) properties are enforceable. Given *any* safety (resp. guarantee, obligation, response) property φ, and a Streett automaton recognizing φ, one can *synthesize* from this automaton an enforcing monitor for φ using systematic transformations. This also proves the correctness of these transformations.

THEOREM 5.1. *Given a program $\mathcal{P}_\Sigma$, a reasonable safety (resp. guarantee, response) property $\varphi \in \text{Safety}_\Sigma$ (resp. $\varphi \in \text{Guarantee}_\Sigma, \varphi \in \text{Obligation}_\Sigma, \varphi \in \text{Response}_\Sigma$) is enforceable on $\mathcal{P}_\Sigma$ by an EM obtained by the application of the safety (resp. guarantee, obligation, response) transformation on the automaton recognizing φ. More formally, given $\mathcal{A}_\varphi$ recognizing φ, the following properties hold:*

$(\varphi \in \text{Safety}_\Sigma \wedge \mathcal{A}_{\downarrow\varphi} = \text{TSaf}(\mathcal{A}_\varphi)) \Rightarrow \mathit{Enf}(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma)$
$(\varphi \in \text{Guarantee}_\Sigma \wedge \mathcal{A}_{\downarrow\varphi} = \text{TGuar}(\mathcal{A}_\varphi)) \Rightarrow \mathit{Enf}(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma)$
$(\varphi \in \text{Obligation}_\Sigma \wedge \mathcal{A}_{\downarrow\varphi} = \text{TOblig}(\mathcal{A}_\varphi)) \Rightarrow \mathit{Enf}(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma)$
$(\varphi \in \text{Response}_\Sigma \wedge \mathcal{A}_{\downarrow\varphi} = \text{TResp}(\mathcal{A}_\varphi)) \Rightarrow \mathit{Enf}(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma)$

5.2.2 Non-enforceable properties.

The persistence class contains non enforceable properties. More precisely, *pure* persistence properties (*i.e.* properties of $\text{Persistence}_\Sigma \setminus \text{Obligation}_\Sigma$) can not be enforced by EMs[2].

EXAMPLE 5.2 (NON-ENFORCEABLE PROPERTY). *For an alphabet $\Sigma \supset \{a\}$, an example of pure persistence property is $\Sigma^* \cdot a^\omega$ stating that "it will be eventually true that a always occurs". This property is recognized by the Streett automaton depicted on Fig. 5 with $\mathit{vinf}(\sigma) \subseteq P, P = \mathit{vinf}(\sigma) = \{1\}$.*

[2]As the safety-progress classification is a hierarchy, the classes of safety, guarantee, and obligation properties are contained in the persistence one. Consequently, some persistence properties are enforceable: properties of the enforceable classes.

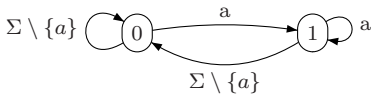

Figure 5: Automaton recognizing the persistence property $\Sigma^* \cdot a^\omega$

One can understand the enforcement limitation intuitively using the following argument: if this property was enforceable it would have implied that an enforcement monitor could decide from a certain point that the underlying program will always produce the event a. However such a decision can never be taken by a monitor without memorizing the entire execution sequence beforehand. This is of course unrealistic for an infinite sequence.

6. COMPARING EMS WITH OTHER RUNTIME ENFORCEMENT MECHANISMS

To the best of the authors knowledge, edit-automata are the most powerful (in terms of enforcement ability) enforcement mechanism. It is worth noticing that the model of enforcement monitor introduced in this paper is confined with the same enforcement aptitude: with edit-automata the enforcement capability is obtained through an infinite set of states, whereas EMs have a finite set of control states extended with a finite set of enforcement actions operating on a (possibly unbounded) memory. However, this work still brings original and interesting results regarding enforcement monitoring.

First, we propose a systematic translation of a *recognizing* automaton into an *enforcing* one. This systematic transformation eases the definition of the enforcement mechanism. Finding, and encoding an enforcement mechanism using edit-automata is not an intuitive operation, and there is usually a gap between the initial property and the associated edit automata. Therefore, ensuring formally that the edit automata obtained enforces the correct property is not an easy task.

Moreover, our enforcement monitor model proposes a clear distinction between control state (used for property recognition) and the memory device (used for sequence memorization). Advantages of such a partitioning are twofold. Firstly, such a mechanism is much closer to implementation issues. Therefore it facilitates the implementation process and makes it more compatible with formal reasoning, providing more confidence. Secondly, this provides genericity to our model. For example it allows to study other memorizing policies (e.g, using a bag instead of a FIFO queue). Varying the memory policies seems to us a good starting point to tackle practical constraints of runtime enforcement monitoring.

7. A PROTOTYPE IMPLEMENTATION

In this section we overview our prototype toolbox (depicted on Fig. 6) which implements the previously defined approach. Our framework is implemented as a Java toolbox, using Aspect Oriented Programming [11] as an underlying technique. Taking, as input, a property φ specified by

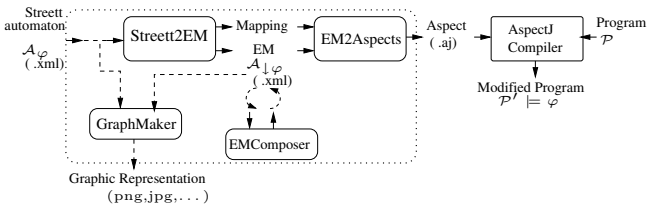

Figure 6: Our prototype toolbox

```
<automaton P="null">
 <alphabet name="sigma">
  <symbol name="a"/><symbol name="b"/><symbol name="c"/>
 </alphabet>

 <state id="1" initial="true"  R="false" >
  <transition nextState="1">
   <event value="a"/>
  </transition>
  <transition nextState="2">
   <event value="b"/> <event value="c"/>
  </transition>
 </state>
 <state id="2" initial="false" R="true">
  <transition nextState="2">
   <event value="b"/>
  </transition>
  <transition nextState="1">
   <event value="a"/>
  </transition>
  <transition nextState="3">
   <event value="c"/>
  </transition>
 </state>
 <state id="3" initial="false" R="false">
  <transition nextState="3">
   <event value="sigma"/>
  </transition>
 </state>
</automaton>
```

Figure 7: Streett automaton of Fig. 4 in XML

a Streett automaton $\mathcal{A}_\varphi$, encoded in XML, it produces an (AspectJ) aspect to be weaved with a target Java program $\mathcal{P}$. The resulting program $\mathcal{P}'$ then meets property φ, in the sense that this property is actually enforced.

On Fig. 7 is represented the Streett automata introduced in Fig. 4 encoded in our XML format. The XML schema we considered consists in two parts: the first one describes the automaton alphabet and the second one gives list of states and outgoing transitions.

7.1 Streett2EM: synthesizing EMs from Streett automata

This tool consists mainly in implementing the aforementioned transformations in Sect. 5.1 (TSaf, Tguar, etc...). Processing an inputted Streett automaton expressed in an XML format, it produces a new XML file representing the corresponding EM. To do so, we have chosen to use an XSLT transformation to implement the transformations. First, the Streett automaton is processed by a parser whose purpose is to check its validity regarding an XSD grammar, and verify its soundness (since we consider only deterministic and complete Streett automata).

Once these preliminary validations steps are completed, the automaton is submitted to an XSLT transformation in order to obtain the expected EM (described in XML). To each synthesis transformation is associated a specific

```
<xsl:if test="$stateP='null' ">
 <xsl:variable name="followingStateR">
  <xsl:call-template name="return_state_R">
   <xsl:with-param name="stateFollowing">
    <xsl:value-of select="@nextState"/>
   </xsl:with-param>
  </xsl:call-template>
 </xsl:variable>
 <!--For guarantee automata-->
 <xsl:if test="$stateR='false' ">
  <xsl:if test="$followingStateR='true' ">
   <xsl:attribute name="value">dump</xsl:attribute>
  </xsl:if>
 </xsl:if>
 <xsl:if test="$stateR='true' ">
  <xsl:if test="$followingStateR='true' ">
   <xsl:attribute name="value">dump</xsl:attribute>
  </xsl:if>
 </xsl:if>
```

Figure 8: Extract of the XSLT transformation used in Streett2EM

XSLT transformation. On Fig. 8 is represented an extract of this XSLT transformation. One can find the first rule for the transformation of transition of a guarantee automaton. From an abstract point of view, for each transition, a *dump* operation is selected if the destination state of the transition belongs to the recurrent states (variable `$followingStateR`).

This tool also generates a mapping skeleton whose purpose is to link abstract events used in the automata (their alphabet), to the corresponding concrete events on the underlying Java program (*e.g.* method call, field access). It consists in a list of event which are retrieved from the inputted Streett automaton. This skeleton should be filled by the user (indicating the list of concrete events associated to each abstract event).

7.2 Composing EMs

Not exposed in this paper, but in the companion report, is the opportunity of composing EMs. When generating EM one may want to compose it as to build more complex EMs. The EMComposer implements general purpose operations of composition (*e.g.* union, intersection). The formal principle is detailled in [8]. Roughly speaking, to perform a binary operation between two inputted EMs, the EMComposer builds a product of the two underlying automata. Then it combines the enforcement operations associated to each EM on a given event. This general method of performing first the underlying automata product, and then combining the enforcement operation according to the semantics of the performed operations is generic. Indeed, one is able to add easily its own operator for EM combination. Adding a new composition operator for EMs amounts to define the composition function $Ops \times Ops \rightarrow Ops$ determining how to combine the enforcement operations of the two operand EMs. This component of the toolbox seems essential to us as it permits the composition of EMs in order to enforce complex properties which may arise from security policies.

7.3 EM2Aspect: synthesizing Aspect from EM

The last tool is used to generate an AspectJ Java aspect from an enforcement monitor. EM2Aspect processes the XML files of the EM and the user-completed mapping by first performing syntactic analysis of these two entities. Then an abstract data structure is generated by picking up information from the EM and the mapping. This datastruc-

ture is then translated into an aspect following this principle: pointcuts are associated to each event of the automata, then advices encode the enforcement operations (*Ops*) performed by the EM on the underlying program upon occurrence of relevant events.

7.4 GraphMaker: graphs for Streett automata and EMs

We add to our toolbox an auxilliary component providing a mean to display a graphic representation of a Streett automaton or an enforcement monitor. The GraphMaker processes an inputted Streett automaton or EM in the XML format previously depicted, and then use GraphViz [2] as an underlying tool for graph generation. As an example, graphs of Fig. 4 are obtained using the GraphMaker component.

8. CONCLUSION AND PERSPECTIVES

In this paper our purpose was to extend previous works on property checking through runtime enforcement in several directions. Firstly, we proposed a generic notion of enforcement monitors based on a memory device, finite sets of control states and enforcement operations. This notion of EM encompasses previous similar ones: security-automata (and consequently shallow-history automata) and edit-automata in a rather obvious way. Moreover, we specified their enforcement abilities wrt. the general safety-progress classification of properties. It allowed a fine-grain characterization of the space of enforceable properties. Also, we proposed a systematic technique to produce an enforcing monitor from the Streett automaton recognizing a given safety, guarantee, obligation or response security property. Finally, this approach was implemented in a prototype toolbox and applied to simple case-studies.

An important working direction is now to make this runtime enforcement technique better able to cope with practical limitations in order to deal with larger examples. In particular it is likely the case that not all events produced by an underlying program could be freely observed, suppressed, or inserted. This leads to well-known notions of *observable* and/or *controlable* events, that have to be taken into account by the enforcement mechanisms. Moreover, it could be also necessary to limit the resources consumed by the monitor by storing in memory only an *abstraction* of the sequence of events observed (i.e. using a *bag* instead of a FIFO queue). From a theoretical point of view, this means to define enforcement up to some *abstraction preserving trace equivalence relations*. We strongly believe that our notion of enforcement monitors (with a generic memory device) is a suitable framework to study and implement these features.

Another current working direction is to further extend the prototype tool. Indeed, it will be a good plateform to investigate the impact of the aforementioned practical constraints. Also, we are currently studying alternative rewriting techniques (non based on aspects) to replace the EM2Aspect tool (such as BCEL [19] technology, or dynamic binary code insertion [16]). The benefits would be to perform runtime enforcement from binary versions of the target program.

9. ACKNOWLEDMENTS

Our thanks to the referees for their helpful remarks.

10. REFERENCES

[1] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Cornell University, Ithaca, NY, USA, 1984.

[2] AT&T Research. Graph Visualization Software. http://www.graphviz.org, 2007.

[3] E. Chang, Z. Manna, and A. Pnueli. The safety-progress classification. Technical report, Stanford University, Dept. of Computer Science, 1992.

[4] E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Automata, Languages and Programming*, pages 474–486, 1992.

[5] Erlingsson and Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[6] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004. Adviser-Fred B. Schneider.

[7] U. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *In IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[8] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties. Technical Report TR-2008-7, Verimag Research Report, 2008.

[9] P. W. L. Fong. Access control by tracking shallow execution history. *sp*, 00:43, 2004.

[10] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[12] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977.

[13] J. Ligatti, L. Bauer, and D. Walker. Composing Expressive Run-time Security Policies. *ACM Transactions on Software Engineering and Methodology*, Nov. 07.

[14] J. Ligatti, L. Bauer, and D. Walker. Runtime Enforcement of Nonsafety Policies. *ACM*, Jan. 07.

[15] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *ESORICS*, pages 355–373, 2005.

[16] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[17] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[18] R. S. Streett. Propositional dynamic logic of looping and converse. In *STOC '81 - ACM symposium on Theory of computing*, pages 375–383, New York, NY, USA, 1981. ACM.

[19] The Apache Jakarta Project. Byte Code Engineering Library. http://jakarta.apache.org/bcel/.

[20] M. Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2000.