Author manuscript, published in "ICISS'08: International Conference on Information Systems Security (2008) 41-55"

hal-00346063, version 1 - 11 Dec 2008

Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties

Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier
Firstname.Lastname@imag.fr

Verimag, Université Grenoble I, INPG, CNRS

Abstract. Runtime enforcement is a powerful technique to ensure that a program will respect a given security policy. 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 w.r.t. the *general* safety-progress classification of properties. It allows a fine-grain characterization of the space of enforceable properties. Finally, we propose a *systematic* technique to produce an enforcement monitor from the Streett automaton recognizing a given safety, guarantee, obligation or response security property.

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. Although this trend clearly improves programmers productivity, it also raises some important security issues. Indeed, each individual component should guarantee some individual properties at run-time to ensure that the whole application will respect some given security policy. 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.

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 observe 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 leave any original execution sequence unchanged when already correct, or output its *longest correct prefix* otherwise.

hal-00346063, version 1 - 11 Dec 2008

Runtime enforcement monitoring was initiated by the work of Schneider [1] 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 [2], stating that *something bad can never happen*. Later, Viswanathan [3] noticed that the class of enforceable properties is impacted by the computational power of the enforcement monitor: since the enforcement mechanism cannot implement more than computable functions, only decidable properties can be enforced. More recently [4, 5], 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., froze) and "insert" (frozen) actions in the current execution sequence. To better cope with practical resource constraints, Fong [6] 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.

In this paper, we propose to extend these previous works in several directions. Firstly, we study the enforcement capabilities relatively to the so-called *safety-progress* hierarchy of properties [7, 8]. This classification differs from the more classical safety-liveness classification [9, 10] by offering a rather clear characterization of a number of interesting kinds of properties (*e.g.* obligation, accessibility, justice, etc.), particularly relevant in the security context. Moreover this classification features properties, such as *transactional properties* (i.e, an action pattern to be repeated infinitely), which are neither safety nor liveness properties. Thus, using this classification as a basis 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 enforcement monitor encompasses the previous notions of security automata, edit-automata and "shallow history" automata. The companion report [11] exposes more details and complete proofs of the theorems of this paper.

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 safety-progress classification. Sect. 6 discusses some implementation issues. Finally, the Sect. 7 exposes some concluding remarks.

hal-00346063, version 1 - 11 Dec 2008

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

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 $\sigma : \mathbb{N} \rightarrow E$ 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 σ' when $\forall i \in \{0, \ldots, |\sigma| - 1\} \cdot \sigma_i = \sigma'_i$. When $\sigma' \in E^*$, we note $\sigma \preceq \sigma' \stackrel{\text{def}}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$. For $\sigma \in E^\infty$, we will need to designate its subsequences.

Execution sequences. 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. 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 $Exec(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$. This set is *prefix-closed*, that is $\forall \sigma \in Exec(\mathcal{P}_\Sigma), \sigma' \in \Sigma^* \cdot \sigma' \preceq \sigma \Rightarrow \sigma' \in Exec(\mathcal{P}_\Sigma)$.

Such execution sequences can be made of access events on a secure system to its ressources, or kernel operations on an operating system. In a software context, these events may be abstractions of relevant instructions such as variable modifications or procedure calls.

2.2 Properties

Properties as sets of of execution sequences. In this paper we aim to enforce properties on program. 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 is that properties we will consider are restricted to *single* execution sequences, excluding specific properties defined on powersets of execution sequences.

hal-00346063, version 1 - 11 Dec 2008

Reasonable properties. As noticed in [4], a property can be effectively enforced at runtime only if it is *reasonable* in the following sense: it should be satisfied by the empty sequence, and it should be decidable.This means that a program which performs no action should not violate this property, and that deciding whether any finite execution sequence satisfies or not this property should be a computable function ([3]).

3 A Safety-Progress classification of properties

This section recalls and extends some results about the safety-progress [7, 8, 12] classification of properties. In the original papers this classification introduced a hierarchy between properties defined as *infinite* execution sequences. We extend here this classification to deal also with finite-length execution sequences.

3.1 Generalities about the classification

The safety-progress classification is constituted of four basic classes defined over infinite execution sequences:

- *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* is the class obtained by positive boolean combinations of safety and guarantee properties.
- The *reactivity class* is the class obtained by boolean combinations of response and persistence properties.

Example 1. Let consider an operating system where a given operation *op* is allowed only when an authorization *auth* has been granted before. Then,

- the property φ_1 stating that "an authorization grant *grant_auth* should precede any occurrence of *op*" is a *safety* property;
- the property φ_2 stating that "the next authorization request *req_auth* should be eventually followed by a grant (*grant_auth*) or a deny (*deny_auth*)" is a *guarantee* property;
- the property φ_3 stating that "the system should run forever, unless a *deny_auth* is issued, meaning that every user should be disconnected and the system should terminate" is an *obligation* property;

hal-00346063, version 1 - 11 Dec 2008

– the property φ_4 stating that "each occurrence of *req_auth* should be first written in a log file and then answered either with a *grant_auth* or a *deny_auth* without any occurrence of *op* in the meantime" is a *response* property;
– the property φ_5 stating that "an incorrect use of operation *op* should imply that any future call to *req_auth* will always result in a *deny_auth* answer" is a *persistence* property.

The safety-progress classification is an alternative to the more classical safety-liveness [9, 10] dichotomy. Unlike this later, the safety-progress classification is a hierarchy and not a partition. It provides a finer-grain classification, and the properties of each class can be characterized according to four *views* [7]. We shall consider here only the so-called *automata view*.

3.2 The automata view

First, we define a notion of *property recognizer* using a variant of deterministic and complete Streett automata (introduced in [13] and used in [7]).

Definition 1 (Streett automaton). *A deterministic Streett automaton is a tuple* $(Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ *defined relatively to a set of events* Σ. *The set* Q *is the set of automaton states, where* $q_{\text{init}} \in Q$ *is the initial state. The total function* $\longrightarrow : Q \times \Sigma \rightarrow Q$ *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* $i \leq n$, $R_i \subseteq Q$ *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 deterministic Streett m-automaton.

The *run* of σ on $\mathcal{A}$ is the sequence of states involved by the execution of σ on $\mathcal{A}$. It is formally defined as $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$ where $\forall i \cdot (q_i \in Q^\mathcal{A} \wedge q_i \xrightarrow{\sigma_i}_\mathcal{A} q_{i+1}) \wedge q_0 = q_{\text{init}}{}^\mathcal{A}$. The *trace* resulting in the execution of σ on $\mathcal{A}$ is the unique sequence (finite or not) of tuples $(q_0, \sigma_0, q_1) \cdot (q_1, \sigma_1, q_2) \cdots$ where $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$. We denote by $vinf(\sigma, \mathcal{A})$ (or $vinf(\sigma)$ when clear from context) the set of states appearing infinitely often in $run(\sigma, \mathcal{A})$. This set is formally defined as follows: $vinf(\sigma, \mathcal{A}) = \{q \in Q^\mathcal{A} \mid \forall n \in \mathbb{N}, \exists m \in \mathbb{N} \cdot m > n \wedge q = q_m$ with $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$.

The following definition tells whether an execution sequence is accepted or not by a Streett automaton:

Definition 2 (Acceptance condition of a Street automaton). *For an* infinite *execution sequence* $\sigma \in \Sigma^\omega$, *we say that* $\mathcal{A}$ *accepts* σ *if* $\forall i \in \{1, \ldots, m\} \cdot vinf(\sigma, \mathcal{A}) \cap R_i \neq \emptyset \vee vinf(\sigma, \mathcal{A}) \subseteq P_i$.

hal-00346063, version 1 - 11 Dec 2008

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

Note that this definition of acceptance condition for finite sequences matches the definition of finitary properties defined in [7] (see [11] for more details).

The hierarchy of automata. Each class of the safety-progress classification is characterized by setting syntactic restrictions on a deterministic Streett 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 $q' \in \overline{R}$.
- An *m-obligation automaton* is an m-automaton such that for each i in $\{1, \ldots, m\}$:
  - there is no transition from $q \in \overline{P_i}$ to $q' \in P_i$,
  - there is no transition from $q \in R_i$ to $q' \in \overline{R_i}$,
- A *response automaton* is a plain automaton such that $P = \emptyset$
- A *persistence automaton* is a plain automaton such that $R = \emptyset$.
- A *m-reactivity automaton* is any unrestricted m-automaton.

Automata and properties. We say that a Streett automaton $\mathcal{A}_\varphi$ *defines* a property φ (defined as a subset of Σ^∞) 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. A property φ that is specifiable by an automaton is a κ-property iff the automaton is a κ-automaton, where $\kappa \in \{\text{safety}, \text{guarantee}, \text{obligation}, \text{response}, \text{persistence}, \text{reactivity}\}$. In the following we note Safety(Σ) (resp. Guarantee(Σ), Obligation(Σ), Response(Σ), Persistence(Σ), Reactivity(Σ)) the set of safety (resp. guarantee, obligation, response, persistence, reactivity) properties defined over Σ.

4 Property enforcement via 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 of $\mathcal{P}$ 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.

hal-00346063, version 1 - 11 Dec 2008

4.1 Enforcement monitors

We define now the central notion of enforcement monitor. Such a runtime device monitors a target program by observing relevant events and performing some enforcement operation depending on its internal state.

Definition 3 (Enforcement monitor (EM)). *An* enforcement monitor $\mathcal{A}_\downarrow$ *is a 4-tuple* $(Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \mathit{Stop}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$ *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* $\mathit{Stop}^{\mathcal{A}_\downarrow}$ *is the set of stopping states (*$\mathit{Stop}^{\mathcal{A}_\downarrow} \subseteq Q^{\mathcal{A}_\downarrow}$*). The partial function (but complete wrt.* $Q^{\mathcal{A}_\downarrow} \times \Sigma$*)* $\longrightarrow_{\mathcal{A}_\downarrow}$*:* $Q^{\mathcal{A}_\downarrow} \times \Sigma \times \mathit{Ops} \rightarrow Q^{\mathcal{A}_\downarrow}$ *is the transition function. In the following we abbreviate* $\longrightarrow_{\mathcal{A}_\downarrow}(q, a, \alpha) = q'$ *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:* $\forall q \in \mathit{Stop}^{\mathcal{A}_\downarrow} \cdot \forall a \in \Sigma \cdot \forall \alpha \in \mathit{Ops} \cdot \forall q' \in Q^{\mathcal{A}_\downarrow} \cdot q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q' \Rightarrow q' \in \mathit{Stop}^{\mathcal{A}_\downarrow}$*.*

Notions of *run* and *trace* (see Sect. 3.2) are naturally transposed from Streett automata. In the remainder of this section, $\sigma \in \Sigma^\infty$ designates an execution sequence of a program, and $\mathcal{A}_\downarrow = (Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \mathit{Stop}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$ designates an EM.

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 comes back to a correct 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 (Enforcement operations Ops). *Enforcement operations take as inputs an event and a memory content (i.e., a sequence of events) to produce a new memory content and an output sequence:* $\mathit{Ops} \subseteq 2^{((\Sigma \cup \{\epsilon\}) \times \Sigma^*) \rightarrow (\Sigma^* \times \Sigma^*)}$*. In the following we consider a set* $\mathit{Ops} = \{\mathit{halt}, \mathit{store}, \mathit{dump}\}$ *defined as follows:* $\forall a \in \Sigma \cup \{\epsilon\}, \forall m \cdot \Sigma^*$

$$\mathit{halt}(a, m) = (\epsilon, m) \qquad \mathit{store}(a, m) = (\epsilon, m.a) \qquad \mathit{dump}(a, m) = (m.a, \epsilon)$$

In the following we assume that outgoing transitions from a stopping state are all labeled with an *halt* operation. That is: $\forall q \in \mathit{Stop}^{\mathcal{A}_\downarrow} \cdot \forall a \in \Sigma \cdot \forall \alpha \in \mathit{Ops} \cdot \forall q' \in Q^{\mathcal{A}_\downarrow} \cdot q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q' \Rightarrow \alpha = \mathit{halt}$.

Definition 5 (EM configurations and derivations). *For an EM* $\mathcal{A}_\downarrow = (Q^{\mathcal{A}_\downarrow}, {q_{\text{init}}}^{\mathcal{A}_\downarrow}, \mathit{Stop}^{\mathcal{A}_\downarrow}, \longrightarrow_{\mathcal{A}_\downarrow})$*, 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* $\sigma = a.\sigma' \wedge q \xrightarrow{a/\alpha}_{\mathcal{A}_\downarrow} q' \wedge \alpha(a, m) = (o, m')$*.*

hal-00346063, version 1 - 11 Dec 2008

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 $o = o_0 \cdot o_1 \cdots o_{k-1}$.*

Besides, 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, namely how it transforms an input sequence σ into an output sequence o by performing derivation steps from its initial state. For the upcoming definitions we will distinguish between finite and infinite sequences and 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 6 (Sequence transformation). *We say that:*

- *The sequence $\sigma \in \Sigma^*$ 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 $\forall \sigma' \in \Sigma^* \cdot \sigma' \prec \sigma \cdot (\exists o' \in \Sigma^* \cdot ({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma') \Downarrow_{\mathcal{A}_\downarrow} o' \wedge o' \preceq o)$.*

Definition 7 (Property-Enforcement). *Let 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*

- $\forall \sigma \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^*, \exists o \in \Sigma^* \cdot enforced(\sigma, o, \mathcal{A}_\downarrow, \varphi)$*, where the predicate $enforced(\sigma, o, \mathcal{A}_\downarrow, \varphi)$ is the conjonction of the following constraints:*

$$({q_{\text{init}}}^{\mathcal{A}_\downarrow}, \sigma) \Downarrow_{\mathcal{A}_\downarrow} o \quad (1)$$

$$\varphi(o) \quad (2)$$

$$\varphi(\sigma) \Rightarrow \sigma = o \quad (3)$$

$$\neg\varphi(\sigma) \Rightarrow \Big(\exists \sigma' \prec \sigma \cdot \varphi(\sigma') \wedge o = \sigma' \wedge \big(\not\exists \sigma'' \succ \sigma' \cdot \varphi(\sigma'') \wedge \sigma'' \preceq \sigma\big)\Big) \quad (4)$$

- $\forall \sigma' \in Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega, \forall \sigma \prec \sigma', \exists o \in \Sigma^* \cdot enforced(\sigma, o, \mathcal{A}_\downarrow, \varphi)$.

(1) *stipulates that the sequence σ is transformed by $\mathcal{A}_\downarrow$ into a sequence o,* (2) *states that o satisfies the property φ,* (3) *ensures transparency of $\mathcal{A}_\downarrow$, i.e. if σ satisfied already the property then it is not transformed, and* (4) *ensures in the case where σ does not satisfy φ that o is the longest prefix of σ satisfying the property.*

Example 2 (Enforcement monitor). We provide on Fig. 1 two EMs to illustrate the enforcement of some of the properties introduced in example 1.

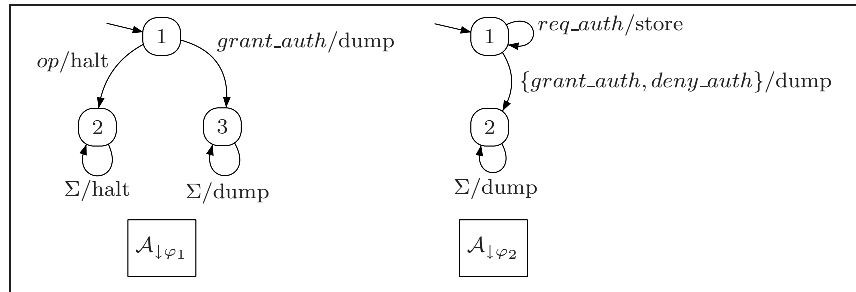

Fig. 1. Two examples of EMs

- The left-hand side of Fig. 1 is an EM $\mathcal{A}_{\downarrow\varphi 1}$ for the safety property φ_1. We assume here that the set Σ of relevant events is $\{op, grant_auth\}$ (other events are ignored by the EM). $\mathcal{A}_{\downarrow\varphi 1}$ has one stopping state, $Stop = \{2\}$, and its initial state is 1. From this initial state it simply *dumps* a first occurrence of *grant_auth* and moves to state 3, where all events of Σ are allowed (i.e., *dumped*). Otherwise, if event *op* occurs first, then it moves to state 2 and halts the underlying program forever.
- The right-hand side of Fig. 1 is an EM $\mathcal{A}_{\downarrow\varphi 2}$ for the guarantee property φ_2. We assume here that the set Σ of relevant events is $\{req_auth, grant_auth, deny_auth\}$. The initial state of $\mathcal{A}_{\downarrow\varphi 2}$ is state 1, and it has no stopping states. Its behavior is the following: occurrences of *req_auth* are *stored* in memory as long as *grant_auth* or *deny_auth* does not occur, then the whole memory content is *dumped*. This ensures that the output sequence always satisfies the property under consideration.

5 Enforcement wrt. the safety-progress 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 EM, and more important, we provide a *systematic construction* of an EM enforcing a property φ from the Streett automaton defining this property. This construction technique is specific to each class of properties.

5.1 From a recognizing automaton to an enforcement monitor

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

Definition 8 (Safety Transformation). *Given a Streett safety-automaton* $\mathcal{A}_\varphi = (Q^{\mathcal{A}_\varphi}, q_{\text{init}}{}^{\mathcal{A}_\varphi}, \Sigma, \longrightarrow_{\mathcal{A}_\varphi}, (\emptyset, P))$ *recognizing a reasonable safety property* φ *defined over a language* Σ*, we define a transformation (named*

hal-00346063, version 1 - 11 Dec 2008

hal-00346063, version 1 - 11 Dec 2008

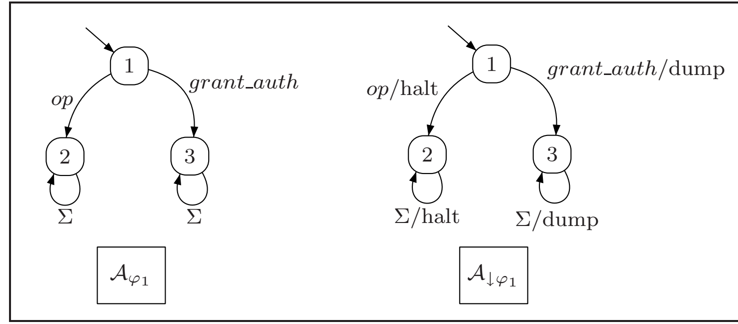

Fig. 2. Recognizing automaton and EM for the safety property φ_1

TransSafety*) of this automaton into an enforcement monitor* $\mathcal{A}_{\downarrow\varphi} = (Q^{\mathcal{A}_{\downarrow\varphi}}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}}, Stop^{\mathcal{A}_{\downarrow\varphi}}, \longrightarrow_{\mathcal{A}_{\downarrow\varphi}})$ *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:*
  - $q \overset{a/dump}{\longrightarrow}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \in P \wedge q \overset{a}{\longrightarrow}_{\mathcal{A}_\varphi} q'$
  - $q \overset{a/halt}{\longrightarrow}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin P \wedge q \overset{a}{\longrightarrow}_{\mathcal{A}_\varphi} q'$

Note that there is no transition (in the Streett automaton) from $q \notin P$ *to* $q' \in P$*, and* $R = \emptyset$*. We note* $\mathcal{A}_{\downarrow\varphi} = \text{TransSafety}(\mathcal{A}_\varphi)$*.*

Informally, the behavior of an EM $\mathcal{A}_{\downarrow\varphi}$ obtained from TransSafety($\mathcal{A}_\varphi$) can be understood as follows. While the current execution sequence satisfies the underlying property (i.e while $\mathcal{A}_\varphi$ remains in P-states), it dumps each input event. Once the execution sequence deviates from the property (i.e., when $\mathcal{A}_\varphi$ reaches a $\overline{P}$-state), then it halts immediately the underlying program with a *halt* operation. The following example illustrates this principle.

Example 3 (Safety Transformation). Fig. 2 (left-hand side) depicts a Streett automaton defining the safety property φ_1 of example 1. Its set of states is $\{1, 2, 3\}$, the initial state is 1, and we have $R = \emptyset$ and $P = \{1, 3\}$. The right-hand side shows the corresponding EM obtained using transformation TransSafety.

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

Definition 9 (Guarantee Transformation). *Let consider a Streett guarantee-automaton* $\mathcal{A}_\varphi = (Q^{\mathcal{A}_\varphi}, {q_{\text{init}}}^{\mathcal{A}_\varphi}, \Sigma, \longrightarrow_{\mathcal{A}_\varphi}, (R, \emptyset))$ *recognizing a property* $\varphi \in Guarantee(\Sigma)$*. We define a transformation* TransGuarantee *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:*

hal-00346063, version 1 - 11 Dec 2008

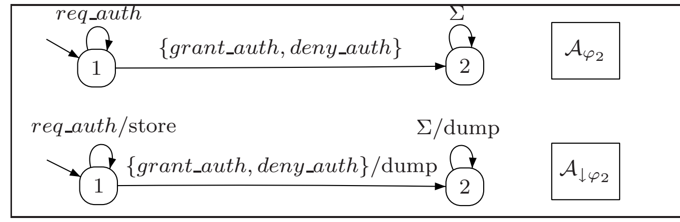

Fig. 3. A guarantee-automaton and the corresponding EM for property φ_2

- $Q^{\mathcal{A}_{\downarrow\varphi}} = Q^{\mathcal{A}_\varphi}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}} = {q_{\text{init}}}^{\mathcal{A}_\varphi}$,
- $Stop^{\mathcal{A}_{\downarrow\varphi}} = \{q \in Q^{\mathcal{A}_{\downarrow\varphi}} \mid \nexists q' \in Reach_{\mathcal{A}_\varphi}(q) \wedge q' \in R\}$
- *the transition relation* $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ *is defined from* $\longrightarrow_{\mathcal{A}_\varphi}$ *as the smallest relation verifying the following rules:*
  - $q \overset{a/dump}{\longrightarrow}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \in R \wedge q \overset{a}{\longrightarrow}_{\mathcal{A}_\varphi} q'$
  - $q \overset{a/halt}{\longrightarrow}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \overset{a}{\longrightarrow}_{\mathcal{A}_\varphi} q' \wedge \nexists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$
  - $q \overset{a/store}{\longrightarrow}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q' \notin R \wedge q \overset{a}{\longrightarrow}_{\mathcal{A}_\varphi} q' \wedge \exists q'' \in R \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$

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* $\mathcal{A}_{\downarrow\varphi} = \text{TransGuarantee}(\mathcal{A}_\varphi)$.

An EM $\mathcal{A}_{\downarrow\varphi}$ obtained from TransGuarantee($\mathcal{A}_\varphi$) behaves as follows. While the current execution sequence does not satisfy the underlying property (i.e., while $\mathcal{A}_\varphi$ remains in $\overline{R}$-states), it stores each entered event in its memory. Once, the execution sequence satisfies the property (i.e., when $\mathcal{A}_\varphi$ reaches an R-state), it dumps the content of the memory and the current event. The following example illustrates this principle.

Example 4 (Guarantee Transformation). Fig. 3 (up) shows a Streett automaton recognizing the guarantee property φ_2 of example 1. Its set of states is $\{1, 2\}$, the initial state is 1, and we have $R = \{2\}$ and $P = \emptyset$. At the bottom is depicted the EM enforcing this same property, obtained by the TransGuarantee transformation. One can notice that this EM has no stopping state.

We now define a transformation for obligation properties. Informally the TransObligation transformation combines the effects of the two previously introduced transformations on using information of each accepting pair.

Definition 10 (Obligation Transformation). *Let consider a Streett obligation-automaton* $\mathcal{A}_\varphi = (Q^{\mathcal{A}_\varphi}, {q_{\text{init}}}^{\mathcal{A}_\varphi}, \Sigma, \longrightarrow_{\mathcal{A}_\varphi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ *recognizing an obligation property* $\varphi \in Obligation(\Sigma)$ *defined over a language* Σ. *We define a transformation* TransObligation *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:*

- $Q^{\mathcal{A}_{\downarrow\varphi}} = Q^{\mathcal{A}_\varphi}, {q_{\text{init}}}^{\mathcal{A}_{\downarrow\varphi}} = {q_{\text{init}}}^{\mathcal{A}_\varphi}$,
- $Stop^{\mathcal{A}_{\downarrow\varphi}} = \bigcup_{i=1}^{m} (\overline{P_i} \bigcap \{q \in Q^{\mathcal{A}_{\downarrow\varphi}} \mid \nexists q' \in Reach_{\mathcal{A}_\varphi}(q) \wedge q' \in R_i\})$

hal-00346063, version 1 - 11 Dec 2008

– *the transition relation* $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ *is defined from* $\longrightarrow_{\mathcal{A}_\varphi}$ *as the smallest relation verifying the following rule:*
$q \xrightarrow{a/\alpha}_{\mathcal{A}_{\downarrow\varphi}} q'$ *if* $q \xrightarrow{a}_{\mathcal{A}_\varphi} q'$ *and* $\alpha = \sqcap_{i=1}^{m}(\sqcup(\beta_i, \gamma_i))$ *where*
  - $\sqcap, \sqcup$ *designate respectively the infimum and the supremum with respect to the complete lattice* $(Ops, \sqsubseteq)$*, where* $halt \sqsubseteq store \sqsubseteq dump$ *(*$\sqsubseteq$ *is a total order),*
  - *and the* β_i *and* γ_i *are obtained in the following way:*
    * $\beta_i = dump$ *if* $q' \in P_i$
    * $\beta_i = halt$ *if* $q' \notin P_i$
    * $\gamma_i = dump$ *if* $q' \in R_i$
    * $\gamma_i = halt$ *if* $q' \notin R_i \wedge \nexists q'' \in R_i \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$
    * $\gamma_i = store$ *if* $q' \notin R_i \wedge \exists q'' \in R_i \cdot q'' \in Reach_{\mathcal{A}_\varphi}(q')$

Note that there is no transition from $q \in R_i$ *to* $q' \in \overline{R_i}$*, and no transition from* $q \in \overline{P_i}$ *to* $q' \in P_i$*.*

Finding a transformation for a *response* property φ needs to slightly extend the definition of TransGuarantee 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 TransResponse obtained from the TransGuarantee transformation (Def. 9) by adding a rule to deal with the aforementioned difference.

Definition 11 (Response Transformation). *Let consider a Streett response-automaton* $\mathcal{A}_\varphi = (Q^{\mathcal{A}_\varphi}, q_{\text{init}}{}^{\mathcal{A}_\varphi}, \Sigma, \longrightarrow_{\mathcal{A}_\varphi}, (R, \emptyset))$ *recognizing a response property* $\varphi \in Response(\Sigma)$ *defined over a language* Σ*. We define a transformation* TransResponse *of this automaton into an enforcement monitor* $\mathcal{A}_{\downarrow\varphi} = (Q^{\mathcal{A}_{\downarrow\varphi}}, q_{\text{init}}{}^{\mathcal{A}_{\downarrow\varphi}}, Stop^{\mathcal{A}_{\downarrow\varphi}}, \longrightarrow_{\mathcal{A}_{\downarrow\varphi}})$ *using the transformations of the* TransResponse *transformation and adding the following rules to define* $\rightarrow_{\mathcal{A}_{\downarrow\varphi}}$ *from* $\rightarrow_{\mathcal{A}_\varphi}$*:*

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

An EM $\mathcal{A}_{\downarrow\varphi}$ obtained from TransGuarantee($\mathcal{A}_\varphi$) behaves as follows. 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 of $\mathcal{A}_\varphi$ is in $\overline{R}$), it stores each event of the input sequence. Once, the execution sequence satisfies the property (the current state of $\mathcal{A}_\varphi$ is in R), it dumps the content of the memory and the current event.

Example 5 (Response Transformation). Fig. 4 (left-hand side) shows a Streett automaton recognizing the response property φ_4 introduced in example 1. Its set of states is $\{1, 2, 3, 4\}$, the initial state is 1, and we have $R = \{1\}$ and $P = \emptyset$. The right-hand side shows the EM enforcement the same property, obtained by the TransResponse transformation. One can notice there is one stopping state 4.

hal-00346063, version 1 - 11 Dec 2008

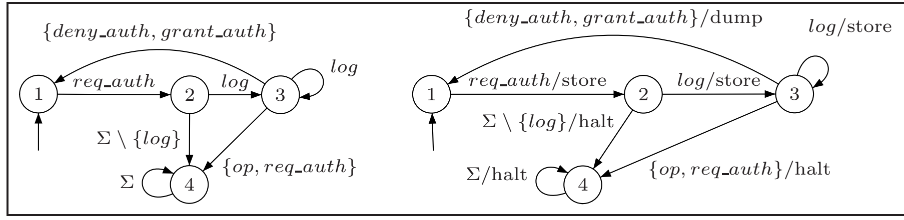

Fig. 4. A response-automaton and the corresponding EM for property φ_4

5.2 Enforcement wrt. the safety-progress 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.

Enforceable properties. Now, we delineate the class of enforceable properties. Notably, 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 1. *Given a program $\mathcal{P}_\Sigma$, a reasonable safety (resp. guarantee, obligation, 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, we have:*

$$\begin{gathered}(\varphi \in \text{Safety}(\Sigma) \wedge \mathcal{A}_{\downarrow\varphi} = \text{TransSafety}(\mathcal{A}_\varphi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma),\\ (\varphi \in \text{Guarantee}(\Sigma) \wedge \mathcal{A}_{\downarrow\varphi} = \text{TransGuarantee}(\mathcal{A}_\varphi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma).\\ (\varphi \in \text{Obligation}(\Sigma) \wedge \mathcal{A}_{\downarrow\varphi} = \text{TransObligation}(\mathcal{A}_\varphi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma).\\ (\varphi \in \text{Response}(\Sigma) \wedge \mathcal{A}_{\downarrow\varphi} = \text{TransResponse}(\mathcal{A}_\varphi)) \Rightarrow Enf(\mathcal{A}_{\downarrow\varphi}, \varphi, \mathcal{P}_\Sigma).\end{gathered}$$

Complete proofs for each class of properties are provided in [11].

Non-enforceable properties. Persistence properties are not enforceable by our enforcement monitors. Such a property is φ_5 introduced in example 1 stating that "an incorrect use of operation *op* should imply that any future call to *req_auth* will always result in a *deny_auth* answer". One can understand the enforcement limitation intuitively using the following argument: if this property was enforceable it would imply that an enforcement monitor could decide from a certain point that the underlying program will always produce the event

hal-00346063, version 1 - 11 Dec 2008

deny_auth in response to a *req_auth*. However such a decision can never be taken without reading and memorizing first the entire execution sequence. This is of course unrealistic for an infinite sequence.

As a straightforward consequence, properties of the reactivity class (containing the persistence class) are not enforceable by our enforcement monitors.

6 Discussion

An important question not yet addressed in this paper concerns the practical issues and possible limitations raised by the approach we propose. These limitations fall in several categories.

First, 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 *controllable* events, that have to be taken into account by the enforcement mechanisms. To illustrate such a limitation, consider a system in which there is a data-flow dependence between two actions. It seems in this case that the enforcement ability is impacted since the first action cannot be frozen (otherwise, the second action could not be executed). Another example is that some actions, important from the security point of view, may not be visible from outside the system (and hence from any enforcement mechanism). Solving these kinds of issues means either further refining the class of enforceable properties (taking these limitations into account), or being able to replace non-controllable or non-observable actions by "equivalent" ones.

Moreover, it could be also necessary to limit the memory resources consumed by the monitor. A possible solution is to store only an *abstraction* of the sequence of events observed (e.g. using a *bag* instead of a FIFO queue, or a set as in [6]). From a theoretical point of view, this means defining 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 this feature.

Furthermore, an other working direction concerns confidence indebted to the implementation of such enforcement monitors. Such an implementation should remain in a size which permits to prove the adequacy between the enforced property and the semantics of the original underlying program. Such a concern follows the well-known principle of minimizing the trusted computing base.

7 Conclusion

In this paper our purpose was to extend in several directions previous works on security checking through runtime enforcement. Firstly, we proposed a generic notion of finite-state enforcement monitors based on generic memory device and enforcement operations. 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, which encompasses previous results on this area. Finally, we proposed a set of (simple) transformations to

hal-00346063, version 1 - 11 Dec 2008

produce an enforcement monitor from the Streett automaton recognizing a given safety, guarantee, obligation or response security property. This feature is particularly appealing, since it allows to automatically generate enforcement monitors from high-level property definitions, like *property specification patterns* [14] commonly used in system verification.

Several perspectives can be foreseen from this work, but the most important issue would be certainly to better demonstrate its practicability, as discussed in Sect. 6. A prototype tool is currently under development, and we plan to evaluate it on relevant case studies in a near future.

References

1. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3** (2000) 30–50
2. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. **28** (2006) 175–205
3. Viswanathan, M.: Foundations for the run-time analysis of software systems. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA (2000) Supervisor-Sampath Kannan and Supervisor-Insup Lee.
4. Jay Ligatti, Lujo Bauer, David Walker: Runtime Enforcement of Nonsafety Policies. ACM (07)
5. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: ESORICS. (2005) 355–373
6. Fong, P.W.L.: Access control by tracking shallow execution history. sp **00** (2004) 43
7. Chang, E., Manna, Z., Pnueli, A.: The safety-progress classification. Technical report, Stanford University, Dept. of Computer Science (1992)
8. Chang, E.Y., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Automata, Languages and Programming. (1992) 474–486
9. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3** (1977) 125–143
10. Alpern, B., Schneider, F.B.: Defining liveness. Technical report, Cornell University, Ithaca, NY, USA (1984)
11. Falcone, Y., Fernandez, J.C., Mounier, L.: Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties. Technical Report TR-2008-7, Verimag Research Report (2008)
12. Manna, Z., Pnueli, A.: A hierarchy of temporal properties (invited paper, 1989). In: PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1990) 377–410
13. Streett, R.S.: Propositional dynamic logic of looping and converse. In: STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (1981) 375–383
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP '98: Proceedings of the second workshop on Formal methods in software practice, New York, NY, USA, ACM (1998) 7–15