UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

DO YOU REALLY MEAN WHAT YOU ACTUALLY ENFORCED?
EDIT AUTOMATA REVISITED

Nataliia Bielova and Fabio Massacci

October 2008

Technical Report # DISI-08-060

Do you really mean what you actually enforced? *

Edit Automata revisited

Nataliia Bielova and Fabio Massacci

DISI - University of Trento,
via Sommarive 14, 38050 Povo, Trento, Italy
surname@disi.unitn.it

Abstract. In the landmark paper on the theoretical side of Polymer, Ligatti and his co-authors have identified a new class of enforcement mechanisms based on the notion of edit automata, that can transform sequences and enforce more than simple safety properties.
We show that there is a gap between the edit automata that one can possibly write (e.g. by Ligatti et al in their running example) and the edit automata that are actually constructed according the theorems from Ligatti's IJIS paper and IC follow-up paper by Talhi et al. "Ligatti's automata" are just a particular kind of edit automata.
Thus, we re-open a question which seemed to have received a definitive answer: you have written your security enforcement mechanism (aka your edit automata); does it really enforce the security policy you wanted?

Key words: Formal models for security, trust and reputation, Resource and Access Control, Validation/Analysis tools and techniques

1 Introduction

The explosion of multi-player games, P2P applications, collaborative tools on Web 2.0, and corporate clients in service oriented architectures has changed the usage models of the average PC user: users demand to install more and more applications from a variety of sources. Unfortunately, the full usage of those applications is at odds with the current security model.

The first hurdle is certification. Certified application by trusted parties can run with full powers while untrusted ones essentially without any powers. However, certification just says that the code is trusted rather than trustworthy because the certificate has no semantics whatsoever. Will your apparently innocuous application collect your private information and upload it to the remote server [19]? Will your corporate client developed in out-sourcing dump your hard disk in a shady country? You have no way to know.

Model carrying code [21] or Security-by-Contract [4] which claim that code should come equipped with a security claims to be matched against the platform policies could be a solution. However this will only be a solution for certified code.

* Research partly supported by the Project EU-FP7-IP-MASTER

To deal with the untrusted code either .NET [12] or Java [7] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous functionalities, such as starting various types of connections or accessing sensitive information. The drawback is that after assigning a permission the user has very limited control over its usage. An application with a permission to upload a video can then send hundreds of them invisibly for the user (see the Blogs on UK Channel 4's Video on Demand application). Conditional permissions that allow and forbid use of the functionality depending on such factors as the bandwidth or some previous actions of the application itself are currently out of reach. The consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted and then they can do almost everything.

To overcome these drawbacks a number of authors have proposed to enforce the compliance of the application to the user's policies by execution monitoring. This is the idea behind security automata [5, 8, 1, 20], safety control of Java programs using temporal logic specs [10] and history based access control [11].

In order to provide enforcement of security policies at run time by monitoring untrusted programs we want to know what kind of policies are enforceable and what sorts of mechanisms can actually enforce them. In a landmark paper [2] Bauer, Ligatti and Walker seemed to provide a definitive answer by presenting a new hierarchy of enforcement mechanisms and classification of security policies that are enforceable by these mechanisms.

Traditional *security automata* were essentially action observers that stopped the execution as soon as an illegal sequence of actions was on the eve of being performed. The new classification of enforcement mechanisms proposed by Ligatti included *truncation, insertion, suppression* and *edit automata* which were considered as execution transformers rather than execution recognizers. The great novelty of these automata was their ability to transform the "bad" program executions in good ones.

These automata were then classified with respect to the properties they can enforce: precisely and effectively enforceable properties. It is stated in [2] that as precise enforcers, edit automata have the same power as truncation, suppression and insertion automata. As for effective enforcement, it is said that edit automata can insert and suppress actions by defining *suppression-rewrite* and *insertion-rewrite* functions and thus can actually enforce more expressive properties than simple safety properties. The proof of Thm. 8 in [2] provides us with a construction of an edit automaton that can effectively enforce any (enforceable) property.

Talhi et al. [22] have further refined the notion by considering bounded version of enforceable properties.

1.1 Contribution of the Paper

If everything is settled why we need to write this paper? Everything started when we tried to formally show "as an exercise" that the running example of edit automaton from [2] provably enforces the security policy described in that

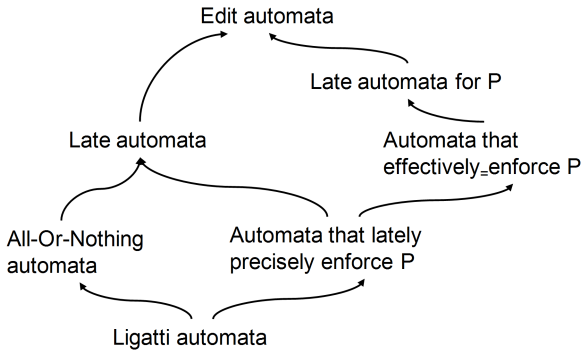

Fig. 1. Relation between different classes of edit automata

paper by applying the effective enforcement theorem from the very same paper. Much to our dismay, we failed.

As a result of this failure we decided to plunge into a deeper investigation and discovered that this was not for lack of will, patience or technique. Rather, the impossibility of reconciling the running example of a paper with the theorem on the very same paper is a consequence of a gap between the edit automata that one can possibly write (e.g. by Ligatti himself in his running example) and the edit automata that are actually constructed according Thm. 8 from [2] and Thm. 8 from [13] and the follow-up papers by Talhi et al. [22]. "Ligatti's automata" are just a particular kind of edit automata.

In a nutshell, we introduce a notion of *Late automata*, a particular kind of edit automata that always outputs some prefix of the input. In Fig. 1 we show the relation between different classes of automata we are investigating in this paper. *All-Or-Nothing automata* at every step outputs the whole input sequence or suppresses the current action. The notion of effective$_=$enforcement is taken from [2].

Fig. 9 later in the paper shows the relations among different classes of edit automata, even though they are the "same" according to [2].

The contribution of this paper is therefore manyfold:

- We show the difference between the running example from [2] and the edit automata that are constructed according Thm. 8 in the very same paper.
- We introduce a more fine grained classification of edit automata and related security properties and relation between different notions of enforcement.
- We further explain the gap by showing that the particular automata that are actually constructed according Thm. 8 from [2] are a particular form of late automata that have an all-or-nothing behavior and that we named Ligatti's automata.
- We show that the construction from Talhi et al. [22] only applies to Ligatti's automata and therefore provide a more useful construction that is the inverse of Talhi et al. [22] construction: namely from a policy specification

expressed as an automaton we show how to construct a Ligatti's automaton that enforces it.

The remainder of the paper is structured as follows. At first we sketch the difference between the edit automaton from the running example and Thm. 8 from [2] (§2). Then we present the basic notions of policies, enforcement and automata in Section 3. We give a more fine grained classification of edit automata introducing the notion of *Late automata* (§4). Section 5 explains relation between different notions of enforcement and types of edit automata. We provide the construction of Ligatti's automaton that enforces a policy expressed as a Policy automaton(§6). Finally we conclude with a discussion of future and related works (§7).

2 The example revised

Example 1 (Verbatim from [2]). To make our example more concrete, we will model a simple market system with two main actions, `take(n)` and `pay(n)`, which represent acquisition of *n* apples and the corresponding payment. We let a range over all the actions that might occur in the system (such as `take`, `pay`, `window-shop`, `browse`, etc.) Our policy is that every time an agent takes *n* apples it must pay for those apples. Payments may come before acquisition or vice versa, and `take(n); pay(n)` is semantically equivalent to `pay(n); take(n)`. The edit automaton enforces the atomicity of this transaction by emitting `take(n); pay(n)` only when the transaction completes. If payment is made first, the automaton allows clients to perform other actions such as `browse` before committing (the `take-pay` transaction appears atomically after all such intermediary actions). On the other hand, if apples are taken and not paid for immediately, we issue a warning and abort the transaction. Consistency is ensured by remembering the number of apples taken or the size of the prepayment in the state of the machine. Once acquisition and payment occur, the sale is final and there are no refunds (durability).

The edit automaton proposed in [2] is shown in Fig. 2. The nodes in the picture represent the automaton states, and the arcs represent the transitions. The action above the arc defines the input action and the sequence below the arc represents the output actions. Arcs with no sequence beneath represent suppression transitions. If there is no arc for the current action, then the automaton halts.

As authors of [2] say, given edit automaton effectively enforces the market policy. But definition of policy presumes that there exists a predicate $\widehat{P}$ over all finite and infinite executions [2]. Then they define notion of *property*, which is a computable predicate over only finite sequences of executions. The definition of effective enforcement includes this notion of property $\widehat{P}$ but the computable predicate over the sequences of actions is not given explicitly in [2] for Ex. 1.

That is why following the example in English we are presenting a property $\widehat{P}$ for the market policy in the Tab. 1. Assuming that our presentation of the policy corresponds to the Ex. 1, the given edit automaton [2] should effectively enforce the policy in Tab. 1. In order to formally define the allowed and prohibited behavior described in the market policy we present: 1) a predicate $\widehat{P}$ over

Table 1. Sequences of actions for market policy

Suspended decision

No	Sequence of actions σ	Expected output	$\widehat{P}(\sigma)$	Decision finalized
1	take(1)	·	×	
2	pay(1)	·	×	
3	pay(1); browse	browse	×	

Clear accept

No	Sequence of actions σ	Expected output	$\widehat{P}(\sigma)$	Decision finalized
4	pay(1); take(1)	pay(1); take(1)	$\surd$	$\surd$

Clear violations

No	Sequence of actions σ	Expected output	$\widehat{P}(\sigma)$	Decision finalized
5	take(1); browse	warn; browse	×	$\surd$
6	take(1); browse; pay(2)	warn; browse	×	
7	take(1); browse; pay(2); take(2)	warn; browse; pay(2); take(2)	×	$\surd$
8	take(1); browse; pay(1)	warn; browse	×	
9	take(1); browse; pay(1); take(2)	warn; browse	×	
10	take(1); browse; pay(1); take(2); browse	warn; browse; warn; browse	×	
11	take(1); browse; pay(1); take(2); browse; pay(2)	warn; browse; warn; browse	×	
12	take(1); pay(2); take(2)	pay(2); take(2)	×	

Unclear cases

No	Sequence of actions σ	Expected output	$\widehat{P}(\sigma)$	Decision finalized
13	pay(1); browse; pay(2)	browse	×	
14	pay(1); browse; pay(2); take(2)	browse; pay(2); take(2)	×	$\surd$(?)
15	pay(1); browse; pay(2); take(2); browse	browse; pay(2); take(2); browse	×	$\surd$(?)

sequences of executions; 2) the expected output for every input sequence according the original example in Tab. 1. In addition, in column "Decision finalized" we mark the sequences in which the decision about the output can be finalized and unmark those, where additional input may be required to make a decision. We may wait for additional input because in some cases it is unclear from the text in English whether the decision can be made or not.

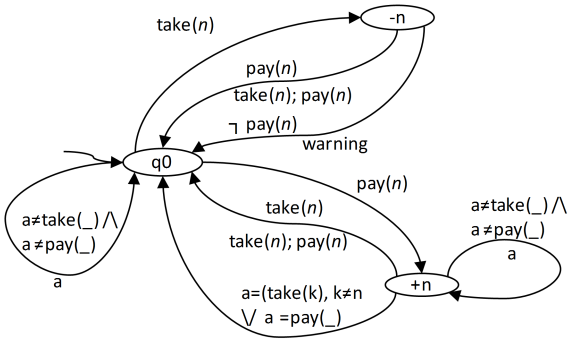

Fig. 2. An edit automaton that "effectively" enforces the market policy [2].

For example let us explain the expected output of sequence 7:

- It contains `take(1)` action and `browse` action after it.
- Since there is no `pay(1)` action, the policy is violated. We expect the action `take(1)` to be suppressed and output the warning action `warn` instead.
- The `browse` action does not violate the policy hence we output it.
- Actions `pay(2)`; `take(2)` do not violate the policy.
- Therefore the output is `warn`; `browse`; `pay(2)`; `take(2)`.

The sequences 1, 2 and 3 are grouped in the "Suspended decision" part of the Tab. 1 due to the fact that decision can be made only when some more actions appear in the input.

The sequence 4 is in the "Clear accept" part of the table because only this sequence is clearly accepted by the given security policy (the `take-pay` transaction is finalized).

We combine the sequences where the `pay(n)` action does not appear immediately after the `take(n)` action in the "Clear violations" part.

We separate sequences 13, 14 and 15 in a group "Unclear cases" because the text of original example leaves opened a number of interpretations. It is clear that good sequences must have a pair of `take(n)` and `pay(n)` as the text implies, but it is not clear whether we allow interleaving of `pay(n)` and `pay(m)`. The text seems to imply that this is not possible.

We say that expected output is defined if either a `take-pay` transaction is completed (after the last `pay(n)` action there is a `take(n)` action) or the transaction is violated (after `take(n)` action there is an action different from `pay(n)`). But it is not clear how the sequence 14 should be transformed because we don't know if the `pay(1)` action should still be followed by the `take(1)` action or it should be simply suppressed.

According to the Thm. 8 of [2], any property $\widehat{P}$ can be effectively enforced by some edit automaton. We will construct such edit automaton following the proof of this theorem.

For the sake of simplicity we build an edit automaton only partly (see Fig. 3) where the action set Σ is $\{\texttt{take(1)}, \texttt{take(2)}, \texttt{pay(1)}, \texttt{pay(2)}, \texttt{browse}\}$. Here we use

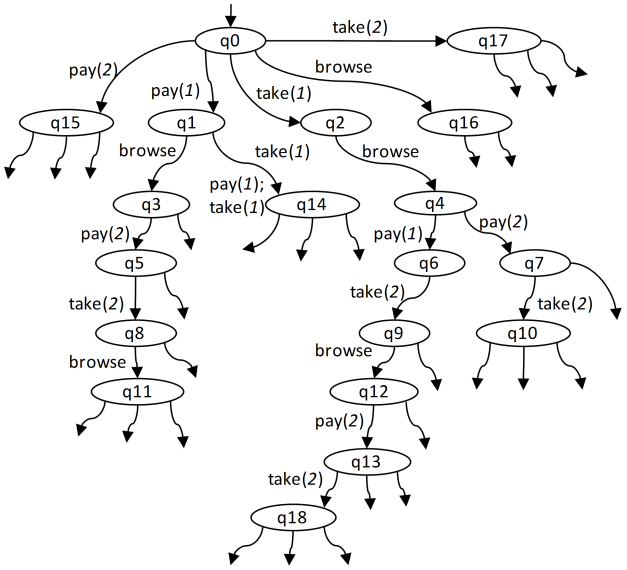

Fig. 3. Constructed edit automaton for the market problem (Ex. 1)

a `browse` action just to present some other actions that the user can do after paying before taking the apples. According to the text of the example, an action `warn` is considered to be an output action in the edit automaton presented by the authors of [2] (see Fig. 2). On the other hand, the `warn` action is not in the set of possible input actions and hence it cannot appear in the output of automaton constructed following the algorithm given as a proof of Thm. 8 [2].

As the cardinality of input language is 5, every state will have five outcoming arcs for all possible actions. We will present here only some of them in order to let the reader see the output sequences for particular input sequences. Constructed automaton is shown in Fig. 3.

As proposed in [2] the state of the constructed automaton is $q \in \Sigma^* \times \Sigma^* \times \{+, -\}$. We denote a state as $\langle \sigma_A, \sigma_S, \{+/-\} \rangle$, where σ_A is the sequence of actions seen so far, σ_S is the sequence of actions seen but not emitted and + (-) is used to indicate that the automaton must not (must) suppress the current action; the notation of Σ^* denotes the set of all finite-length sequences of actions on a system with action set Σ. For example, let us have a look at the the finite input sequence 15: `pay(1)`; `browse`; `pay(2)`; `take(2)`; `browse`. The correspondent trace is $q0, q1, q3, q5, q8, q11$:

- The initial state $q0 = \langle \cdot, \cdot, + \rangle$.
- When the action `pay(1)` is proceeding in state $q0$, it is suppressed because $\neg\widehat{P}(\texttt{pay(1)})$, so the next state is $q1 = \langle \texttt{pay(1)}, \texttt{pay(1)}, + \rangle$.

Table 2. Difference in output for edit automata

No	Input	Output	
		Edit automaton from Fig. 2 [2]	Constructed edit automaton by Thm. 8 [2]
4	pay(1); take(1)	take(1); pay(1)	pay(1); take(1)
11	take(1); browse; pay(1); take(2); browse; pay(2)	warn; browse	·
12	take(1); pay(2); take(2)	warn	·
7	take(1); browse; pay(2); take(2)	warn; take(2); pay(2)	·
15	pay(1); browse; pay(2); take(2); browse	browse; warn	·

- At the next step the action `browse` is proceeding, since $\neg\widehat{P}(\texttt{pay(1)};\texttt{browse})$, this action is also suppressed and next state is $q3 = \langle \texttt{pay(1)};\texttt{browse},\texttt{pay(1)};\texttt{browse}, +\rangle$.
- At every step the action is suppressed according to the predicate $\widehat{P}$. The next state is $q5 = \langle \texttt{pay(1)};\texttt{browse};\ \texttt{pay(2)},\texttt{pay(1)};\ \texttt{browse};\ \texttt{pay(2)},\ +\rangle$.
- The following state is $q8 = \langle \texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)},\texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)},\ +\rangle$.
- $q11 = \langle \texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)};\texttt{browse},\texttt{pay(1)};\ \texttt{browse};\texttt{pay(2)};\texttt{take(2)};\texttt{browse}, +\rangle$

After construction we discovered that edit automaton that effectively enforces $\widehat{P}$ (Fig. 2) and the one constructed by the proof of Thm. 8 (Fig. 3) produce different output for the same input. Let us show in Tab. 2 some cases of input and output of both automata.

Analyzing the Tab. 2 we find out that the transformed sequences of actions are not always the ones expected from the edit automaton. So the question arises: *Why the output is predictable in some cases and unpredictable in the others?* The answer to this question is:

1. When the input sequence is legal both edit automata produce the expected output (e.g. sequence 4)
2. The edit automaton constructed following the proof of Thm. 8 [2] is a very particular kind of the edit automaton.

When the sequence is illegal the output of both edit automata is unexpected. In Fig. 4 we show the relation between input and output for edit automaton from Fig. 2 [2] and edit automata constructed by Thm. 8 [2] with respect to the "good" and "bad" traces. In case of "bad" input sequences edit automaton constructed by Thm. 8 [2] outputs only the longest valid prefix: so either it outputs some valid sequence (sequence 4 in Tab. 2) or suppresses the whole sequence (sequences 11, 12, 7, 15 in Tab. 2). While edit automaton from Fig. 2 [2] always outputs some "good" sequence of actions even if the longest valid prefix is an empty sequence (sequences 11, 12, 7, 15 in Tab. 2).

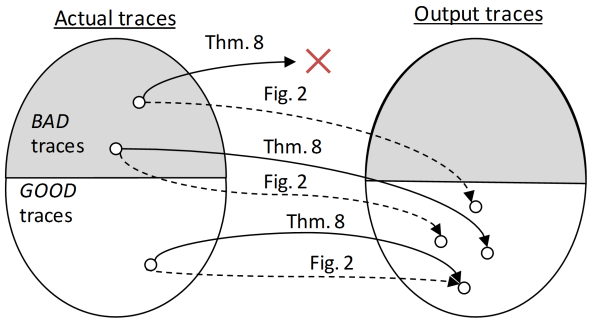

Fig. 4. Relation between input and output for edit automaton from Fig. 2 [2] and edit automaton constructed by Thm. 8 [2]

In order to explain this difference we analyze different classifications of edit automata that explain the behavior of the edit automaton constructed following the proof of Thm. 8 and the edit automaton from Fig. 2 [2]. For example, all theorems referring to edit automata in [22] are about the particular kind of automata that are constructed following the proof of Thm. 8 [2].

3 Basic notions of policies, enforcement and automata

Similarly to [2] we specify the system at a high level of abstraction, where the set Σ is the set of program actions; the set of all finite sequences over Σ is denoted by Σ^*, similarly the set of all infinite sequences is Σ^ω. Execution σ is a finite sequence of actions $a_1, a_2, ..., a_n$.

With $\cdot$ we denote an empty execution. The notation $\sigma[i]$ is used to denote the $(i{+}1)$-th action in the sequence (begin counting at 0). The notation $\sigma[..i]$ denotes the subsequence of σ involving the actions $\sigma[0]$ through $\sigma[i]$, and $\sigma[i{+}1..]$ denotes the subsequence of σ involving all other actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences.

As showed in Section 2 the constructed edit automaton following the algorithm in [2] and edit automaton presented in the same paper are different. We give the original definition of edit automata from [2]:

An *edit automaton* E is described by a 5-tuple of the form $\langle Q, q_0, \delta, \gamma, \omega\rangle$ with respect to some system with actions set Σ. Q specifies possible states, and q_0 is the initial state. The partial function $\delta : (\Sigma \times Q) \rightarrow Q$ specifies the transition function; the partial function $\omega : (\Sigma \times Q) \rightarrow \{-, +\}$ has the same domain as δ and indicates whether or not the action is to be suppressed (-) or emitted (+); the partial function γ is an insertion function, $\gamma : (\Sigma \times Q) \rightarrow \Sigma^* \times Q$. The partial functions δ and γ have disjoint domains.

$$\boxed{(\sigma, q) \xrightarrow{\tau}_E (\sigma', q')} \tag{1}$$

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \text{ if } \sigma = a; \sigma' \wedge \delta(a, q) = q' \wedge \omega(a, q) = + \tag{2}$$

$$(\sigma, q) \xrightarrow{\cdot}_E (\sigma', q') \text{ if } \sigma = a; \sigma' \wedge \delta(a, q) = q' \wedge \omega(a, q) = - \tag{3}$$

$$(\sigma, q) \xrightarrow{\tau}_E (\sigma, q') \text{ if } \sigma = a; \sigma' \wedge \gamma(a, q) = \tau; q' \tag{4}$$

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \text{ otherwise} \tag{5}$$

$$\tag{6}$$

Assuming that the γ function always inserts all necessary actions that have to appear before the a action, we can rewrite the case of insertion as statement (4) and then statement (2). We consider that after inserting some actions τ at the next step the automaton will accept the current action a. Hence, the equation (4) can be represented as follows:

$$(\sigma, q) \xrightarrow{\tau;a}_E (\sigma', q') \text{ if } \sigma = a; \sigma' \wedge \gamma(a, q) = \tau; q' \tag{7}$$

In this way, the sequences σ and σ' are not relevant in the definition of transitions. Loosely speaking this was a Mealy-Moore transformation [9]. In order to give a formal definition in our notations, we will use the σ_S sequence to define the sequence that was read but is not in the output yet.

In [15] the authors provide even more minimal set of rules that are equivalent to the more complex ones above. We give our own definition of edit automaton for better formalization in this paper.

Definition 1 (Edit automata (EA)). *An* edit automaton E *is a 5-tuple of the form $\langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with respect to some system with actions set Σ. Q specifies possible states, and $q_0 \in Q$ is the initial state. The partial function $\delta : (Q \times \Sigma) \rightarrow Q$ specifies the transition function; the partial function $\gamma_o : (Q \times \Sigma \times \Sigma^*) \rightarrow \Sigma^*$ defines the output of the transition according to the current state, the sequence of actions that has been read before the current action and the current input action; the partial function $\gamma_k : (Q \times \Sigma \times \Sigma^*) \rightarrow \Sigma^*$ defined the sequence that will be kept after committing the transition. The dependence between the transition, output and keep function is following: if $\delta(q, a)$ is defined then $\gamma_o(q, a, \sigma)$ and $\gamma_k(q, a, \sigma)$ must be defined for all σ.*

$$\boxed{(q, \sigma_S) \xrightarrow{\gamma_o(q, \sigma_S; a)}_E (q', \gamma_k(q, \sigma_S; a))} \tag{8}$$

In order for the enforcement mechanism to be effective all functions δ, γ_k and γ_o should be decidable.

Definition 2 (Run of an Edit automaton). *Let $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ be an edit automaton. A* run *of A on an* input *sequence of actions $\sigma = \langle a_1, a_2, \ldots \rangle$ is a sequence of pairs $\langle (q_0, \epsilon), (q_1, \sigma_1^k), (q_2, \sigma_2^k), \ldots \rangle$ such that $q_{i+1} = \delta(q_i, a_{i+1})$ and $\sigma_{i+1}^k = \gamma_k(q_i, a_{i+1}, \sigma_i^k)$. The* output *of A on input σ is sequence of actions $\sigma_o = \langle \sigma_1^o, \sigma_2^o, \ldots \rangle$ such that $\sigma_{i+1}^o = \gamma_o(q_i, a_{i+1}, \sigma_i^k)$.*

The example of edit automaton can be found in Fig. 2.

Proposition 1. *The Definition 1 of edit automaton has the same expressive power of the original definition [2].*

Proof. Let us assume that an edit automaton $A = \langle Q, q_0, \quad \delta, \gamma_o, \gamma_k \rangle$ corresponds to the Definition 1 while edit automaton $A^L = \langle Q, q_0, \delta^L, \gamma^L, \omega^L \rangle$ corresponds to the original definition [2]. We have to show that A and A^L have the same expressive power.

For an edit automaton A^L we simply define the keep function as $\gamma_k = \cdot$ and the output function as follows:

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta^L(q,a) = q' \wedge \omega^L(q,a) = +, \\ \cdot & \text{if } \delta^L(q,a) = q' \wedge \omega^L(q,a) = -, \\ \tau; a & \text{if } \gamma^L(a;q) = \tau; q', \\ \cdot & \text{otherwise.} \end{cases} \tag{9}$$

Similarly, for the suppression automaton we define $\gamma_k(q, \sigma_S; a) = \cdot$ and

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta^L(q,a) = q' \wedge \omega^L(q,a) = +, \\ \cdot & \text{if } \delta^L(q,a) = q' \wedge \omega^L(q,a) = -, \\ \cdot & \text{otherwise.} \end{cases} \tag{10}$$

For the insertion automaton $\gamma_k(q, \sigma_S; a) = \cdot$ and

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta^L(q,a) = q', \\ \tau; a & \text{if } \gamma^L(a;q) = \tau; q', \\ \cdot & \text{otherwise.} \end{cases} \tag{11}$$

□

We can decompose the γ_o function $\gamma_o : (\Sigma^* \times Q) \to \Sigma^*$ into function $\gamma'_o : (\Sigma^* \times Q) \to \Sigma^* \times \Sigma$ that defines the output and the next action that will be read by the automaton. In case of insertion automaton reads the action and does not consume it, hence it should read it again. In case of suppression it only consumes it, hence the next action will be taken from the input (we put an empty sequence as a next action in order to show that next action simply has to be taken from the input):

$$\gamma'_o(q, \sigma_S, a) = \begin{cases} (\sigma_o, a) & \text{if } \gamma_o(q, \sigma_S, a) = \sigma_o \wedge \\ & \quad \gamma^L(q,a) = \tau; q', \\ (\sigma_o, \cdot) & \text{if } \gamma_o(q, \sigma_S, a) = \sigma_o \wedge \\ & \quad \delta^L(q,a) = q'. \end{cases} \tag{12}$$

This latter formalization is identical to the one of [2] but more amended to formal treatment of properties. In contrast the definition with the insertion function that always consumes the input action makes it possible to better understand the capabilities of the edit automata as a trace transformer, which is the subject of this paper.

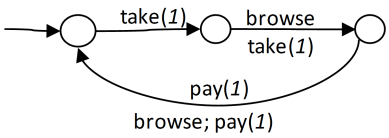

Fig. 5. Example of Late automaton.

4 A new classification of automata

Let us now give a deeper look at the automaton constructed according the proof of Thm. 8 [2]. In this construction at every state the automaton has emitted the sequence σ', and σ' is the *longest valid prefix* of the input sequence σ. Indeed, Tab. 2 shows that this statement holds for the edit automaton constructed by Thm. 8 and it does not hold for the edit automaton from Fig. 2 [2]. Therefore, in order to understand what kind of edit automaton is in Fig. 2 we need to give a formal definition of this kind of automaton. This automaton outputs some valid prefix only when the sequence can become valid again in the future (e.g. for the sequence `take(1)`; `pay(1)`; `take(2)` it will output the valid prefix `take(1)`; `pay(1)`). And it outputs some corrected sequence (current valid prefix and some other sequence) if the sequence cannot become valid in the future(in example 2 of Tab. 2 after reading `take(1)`; `browse` actions the automaton outputs another action `warn`).

This corresponds to the following intuition:

Remark 1. The automaton constructed according to the proof of Thm. 8 in [2] just delays the appearance of input actions until the input has built up a correct sequence again.

Formally, we propose a notion of wider class of such automata called *Late automata*. They simply output some prefix of the input. These class will be the container of other less trivial cases when the property $\widehat{P}$ will be called into account.

Definition 3 (Late automata). Late automaton A *is an edit automaton that is described by a 5-tuple of the form* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the transition is defined as in equation (8) with the restriction that it always outputs some prefix of the input:*

$$\sigma_S; a = \gamma_o(q, \sigma_S; a); \gamma_k(q, \sigma_S; a) \tag{13}$$

The example of Late automaton is shown in Fig. 5. This automaton simply outputs the first action of the input after reading the second and then outputs second and third actions after reading the third action.

In order to give a formal definition of the automata from Thm. 8 [2] for any property $\widehat{P}$ we present also a wider class of automata called *All-Or-Nothing automata*. These automata always output a prefix of the input (hence it is a particular kind of the Late automata). Moreover, at every step of the transition it either outputs all suspended input actions or suppresses the current action.

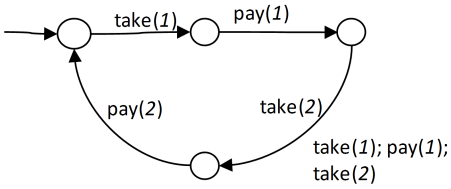

Fig. 6. Example of All-Or-Nothing automaton.

Definition 4 (All-Or-Nothing automata). All-Or-Nothing automaton A *is an edit automaton described by a 5-tuple of the form* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the transition relation is defined as in equation (8) with the following restrictions:*

- *This automaton outputs a prefix of the input: the statement (13) holds.*
- *At every step of the transition either it outputs the whole suspended sequence of actions or suppresses the current action:*

$$\gamma_o(q, \sigma_S; a) = \begin{cases} \sigma_S; a \\ . \end{cases} \tag{14}$$

The example of All-Or-Nothing automaton is given in Fig. 6.

The next step is the refinement of this class towards what we call *Ligatti Automata for* $\widehat{P}$. These automata always output a prefix of the input (hence it is a particular kind of the Late automata) and they are particular kind of All-Or-Nothing automata. Moreover, they output the longest valid prefix. The definition of Ligatti automaton for property $\widehat{P}$ given below was made according to the construction of edit automaton given in the proof of Thm. 8 [2].

Definition 5 (Ligatti automata for property $\widehat{P}$**).** Ligatti automaton E for property $\widehat{P}$ *is an edit automaton described by a 5-tuple of the form* $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the set of states* $Q = \Sigma^*$ *(every state contains the already accepted sequence* σ*) and the transition relation is defined in a similar way as in equation (8):*

$$\boxed{(\sigma, \sigma_S) \overset{\gamma_o(\sigma, \sigma_S; a)}{\longrightarrow}_E (\sigma; \gamma_o(\sigma, \sigma_S; a), \gamma_k(\sigma, \sigma_S; a))} \tag{15}$$

With the following restrictions:

- *The automaton outputs a prefix of the input (the statement (13) holds)*
- *Either it outputs the whole suspended sequence of actions or suppress the current action (the statement (14) holds).*
- *Output is a valid prefix of the input*

$$\widehat{P}(\sigma; \gamma_o(\sigma, \sigma_S; a) \tag{16}$$

- *If the current sequence is valid then it outputs the whole sequence:*

$$\textit{If } \widehat{P}(\sigma; \sigma_S; a) \textit{ then } \gamma_o(\sigma, \sigma_S; a) = \sigma_S; a. \tag{17}$$

At every state a Ligatti automaton for property $\widehat{P}$ keeps the sequence σ that was read till the current moment in order to decide whether $\widehat{P}(\sigma; \sigma_S; a)$ holds. This explains why $Q = \Sigma^*$. In our definition a Ligatti automaton for property $\widehat{P}$ is obviously a particular kind of edit automatfon. We will show that this statement holds in the original definition as well.

Proposition 2. *The Ligatti automaton for property $\widehat{P}$ is an edit automaton according to Ligatti's own Definition.*

Proof. Indeed, given a property $\widehat{P}$ and input sequence σ let us assume that at the current step $a = \sigma[i]$. Then let us define the rewrite functions ω and γ for edit automaton such that Ligatti automaton for property $\widehat{P}$ is an edit automaton. In order to show this we need to keep in the state the sequence of actions σ_A that was already emitted and sequence of actions σ_S that was suppressed in every state: $q = \langle \sigma_A, \sigma_S \rangle$. This is exactly what is done in construction in proof of Thm. 8 [2]. Then we just use the following rewrite functions.

- $\omega(a, q) = +$ if $\widehat{P}(\sigma[..i]) \wedge \sigma_S = \cdot$;
  $\omega(a, q) = -$ if $\neg\widehat{P}(\sigma[..i])$ for suppression, and
- $\gamma(a, q) = \sigma_S, q' \wedge q' = \langle \sigma[..i]; \cdot \rangle$ if $\widehat{P}(\sigma[..i]) \wedge \sigma_S \neq \cdot$ for insertion.

□

Let us now show the inverse of this claim: the edit automaton constructed following the proof of Thm. 8 [2] is a Ligatti Automaton for property $\widehat{P}$.

Proposition 3. *The edit automaton constructed following the proof of Thm. 8 in [2] for property $\widehat{P}$ is a Ligatti automaton for $\widehat{P}$.*

Proof. Consider processing the action a, σ is the input so far, σ_S is a suppressed sequence of actions. Let us have a look at two main steps of construction:

- if $\neg\widehat{P}(\sigma; a)$ then suppress a , $\sigma'_S = \sigma_S; a$.
- if $\widehat{P}(\sigma; a)$ then insert $\sigma_S; a$.

Since at every step the output is $\cdot$ or $\sigma_S; a$ then the automaton obeys the property (14); it always outputs prefix of the input, hence statement (13) holds as well. Constructed automaton outputs the sequence only if it is valid, hence statement (16) holds. It outputs all the suppressed actions if the sequence becomes valid, therefore statement (17) holds as well. Since all the conditions of Ligatti automaton for $\widehat{P}$ are satisfied, we conclude that automaton constructed following the proof of Thm. 8 in [2] for property $\widehat{P}$ is Ligatti automaton for $\widehat{P}$. □

In a nutshell, the difference between edit automata and Ligatti automata for property $\widehat{P}$ is the following:

- edit automata suppress and insert arbitrary actions according to the given rewriting functions ω and γ
- Ligatti automata for property $\widehat{P}$ can only insert those actions that were read before; suppressed actions either will be inserted when the input sequence becomes valid or all subsequent actions will be suppressed.

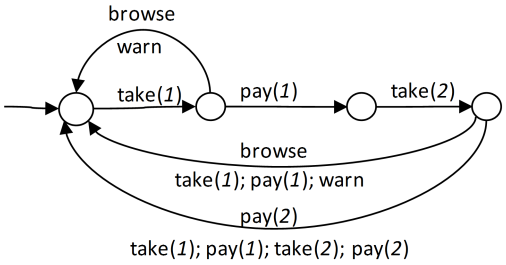

Fig. 7. Example of Late automaton for property $\widehat{P}$.

Therefore Ligatti automata for property $\widehat{P}$ outputs the longest valid prefix of the input sequence.

Since the automaton constructed following the proof of Thm. 8 [2] is a Ligatti automaton for property $\widehat{P}$ while the automaton given in [2] (Fig. 2) is an edit automaton, the difference between their behaviors is not clear.

Still, the automaton of Fig. 2 is not a completely arbitrary edit automaton and we propose a notion of *Late automaton for property* $\widehat{P}$. If the sequence is valid it outputs a valid prefix of the input, otherwise it can output some valid sequence (i.e. fixing the input).

Definition 6 (Late automata for property $\widehat{P}$). Late automaton A for $\widehat{P}$ *is an edit automaton that is described by a 5-tuple of the form* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the transition is defined in the same way as in equation (8) with the following restrictions:*
If $\widehat{P}(\sigma; \sigma_S; a)$ *then*

- *Output is a prefix of the input (the statement (13) holds) and*
- *Output is a valid prefix of the input (the statement (16) holds).*

The example of Late automaton for property $\widehat{P}$ is shown in Fig. 7. This automaton is similar to the one given in Fig. 2 with the only difference that it delays the output of the first `take-pay` transaction.

Later in Fig. 9 we will pictorially describe the relations among different kinds of edit automata. However, in order to explain more relations present in that picture we need first to define the notion of enforcement in the next section.

5 A new classification of enforcement properties

The principles of soundness and transparency were presented in [2] in order to be able to compare different enforcement mechanisms. Let us first see an intuitive description of these mechanisms. The notion of *soundness* requires all the observable output of enforcement mechanism to be valid. The notion of *transparency* means that an enforcement mechanism must preserve the semantics

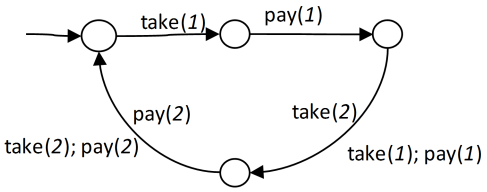

Fig. 8. Example of edit automaton that lately precisely enforces a property $\widehat{P}$.

of executions that are already valid. The notion of *precise* enforcement by [2] obeys both of these properties. According to that definition, the automaton in question outputs program actions in lock-step with the target program's action stream if the action stream σ is valid. Suppose that at the current moment the automaton reads i-th action in the sequence, and the sequence $\sigma[..i+1]$ is not valid. Then the automaton will not output any other actions.

In order to formalize the behavior where the automaton suppresses some actions and later insert them when the sequence turns out to be legal, we present the notion of *Late precise enforcement*.

Definition 7 (Late Precise Enforcement). *An edit automaton A with starting state q_0* lately precisely enforces *a property $\widehat{P}$ on the system with action set Σ iff $\forall \sigma \in \Sigma^* \; \exists q' \; \exists \sigma' \in \Sigma^*$ such that*

1. $(\sigma, q_0) \xrightarrow{\sigma'}{}_A(\cdot, q')$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \wedge \forall i \; \exists j.\; j \leq i \; \exists q_*.\, (\sigma, q_0) \xrightarrow{\sigma[..j]}{}_A(\sigma[i+1..], q_*)$.

Notice that some edit automaton that lately precisely enforces a property $\widehat{P}$ will always output some valid prefix of the input.

We show an example of edit automaton that lately precisely enforces a property $\widehat{P}$ in Fig. 8.

There is another notion of enforcement called "effec-tive$_=$enforcement" [13], which also obeys the properties of soundness and transparency.

Definition 8 (Effective$_=$Enforcement). *An automaton A with starting state q_0* effectively$_=$enforces *a property $\widehat{P}$ on the system with action set Σ iff $\forall \sigma \in \Sigma^* \; \exists q' \quad \exists \sigma' \in \Sigma^*$ such that*

1. $(\sigma, q_0) \xrightarrow{\sigma'}{}_A(\cdot, q')$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$

Let us show the relation between late precise enforcement and effective$_=$enforcement.

Theorem 1. *Edit automata that lately precisely enforce a property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$enforce $\widehat{P}$.*

Proof. At first we have to prove that if edit automaton A lately precisely enforces a property $\widehat{P}$ then it effectively$_=$enforces property $\widehat{P}$.

Since the 1st and 2nd conditions are equal for late precise enforcement and effective enforcement, we have to prove that if the 3rd condition of late precise enforcement holds then the 3rd condition of effective$_=$enfor-cement holds as well, hence we have to prove that if (18) holds then (19) holds as well:

$$\begin{aligned} &\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \wedge \\ &\qquad \wedge \forall i\ \exists j.\ j \leq i\ \exists q_*.\ (\sigma, q_0) \xrightarrow{\sigma[..j]}_A (\sigma[i+1..], q_*) \end{aligned} \tag{18}$$

$$\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \tag{19}$$

We can see that formula (18) is a strengthening of (19). Hence edit automata which lately precisely enforce a property $\widehat{P}$ are a subset of edit automata that effective-ly$_=$enforce $\widehat{P}$.

Next we have to prove that an automaton A^* that effectively$_=$enforces a property $\widehat{P}$ does not lately precisely enforce $\widehat{P}$. In case of illegal input the automaton A^* can output some valid output which is not a prefix of the input, in this case A^* does not lately precisely enforce $\widehat{P}$. □

An example to show that the containment is proper is the edit automaton in Fig. 2 that effectively$_=$enforces property $\widehat{P}$. For an input sequence `take(1); pay(1)` it outputs `take(1); pay(1)` sequence while automaton in Fig. 8 that lately precisely enforces $\widehat{P}$ outputs empty sequence for this input.

Let us come back to Ex. 1. As it is said in [2] the given edit automaton (Fig. 2) effectively$_=$enforces the market policy. But since the market policy is given only in natural language and the predicate $\widehat{P}$ is not given, statements such as "An edit automata effectively enforces the market policy" are a bit stretching the definition.

In Fig. 9 we summarize the relations among the different kind of automata that we have introduced. When drawing two boxes separated by a space we mean that inclusion is probably not proper. In the rest of the paper we prove the correctness of this classification.

Proposition 4. *Late automata and Late automata for property $\widehat{P}$ are not a proper subset of each other.*

Proof. First we have to prove that if edit automaton A is a Late automaton then it is not necessary that A is a Late automaton for property $\widehat{P}$.

By σ we denote an input sequence of the automaton and σ' is an output sequence. A Late automaton A obeys only one property: it always outputs some prefix of the input (statement (13) holds). Hence, even if the overall input sequence σ is valid A can output an invalid prefix of the input ($\neg\widehat{P}(\sigma')$), while Late automaton for property $\widehat{P}$ will always output a valid prefix (statement (16)).

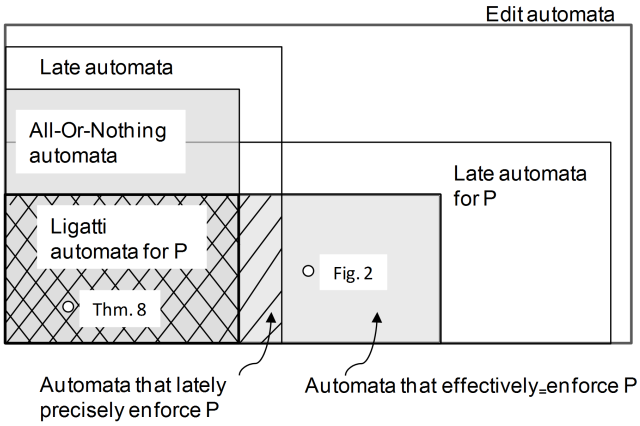

Fig. 9. The classes of edit automata.

Next we have to prove that if edit automaton A is a Late automaton for property $\widehat{P}$ then it is not necessary that A is a Late automaton. In case of invalid input sequence the Late automaton for property $\widehat{P}$ can output another sequence which is not necessarily a prefix of the input, while a Late automaton will always output a prefix of the input (statement 13). □

For example, for input sequence σ = `take(1);browse` the Late automaton from Fig. 5 will output `take(1)` action which is not valid while the Late automaton for property $\widehat{P}$ from Fig. 7 will output `warn` action.

From Proposition 4 we can conclude that classes of Late automata and Late automata for $\widehat{P}$ have some common subclass but none of them include the other.

Theorem 2. *Edit automata that effectively$_=$enforce property $\widehat{P}$ are a proper subset of Late automata for $\widehat{P}$.*

Proof. First we have to prove that if edit automaton A effectively$_=$enforces property $\widehat{P}$ then A is a Late automaton for property $\widehat{P}$.

By σ we denote an input sequence of the automaton and σ' is an output sequence. The automaton A that effectively$_=$enforces $\widehat{P}$ obeys the following properties: a) $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$; b) $\widehat{P}(\sigma')$. Hence it always outputs the valid prefix of the input (the whole sequence in this case), so automaton A is a Late automaton for $\widehat{P}$ according to its definition.

Next we have to prove that if edit automaton A is a Late automaton for property $\widehat{P}$ then it is not necessary that A effectively$_=$enforces property $\widehat{P}$.

In case of a valid input the Late automaton for $\widehat{P}$ will output some valid prefix of the input and not necessary the whole input, hence the property of effective$_=$en-forcement $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$ will not hold. □

For example, given the Late automaton for $\widehat{P}$ from Fig. 7 for a valid input `take(1); pay(1)` it will output an empty sequence while the automaton from

Fig. 2 that effectively$_=$enforces property $\widehat{P}$ for a valid input `take(1); pay(1)` will output `take(1); pay(1)`.

Proposition 5. *Edit automata that effectively$_=$enforce property $\widehat{P}$ are not a subset of Late automata.*

Proof. We have to prove that if an edit automaton A effectively$_=$enforces property $\widehat{P}$ then it is not necessary that A is a Late automaton.

By σ we denote an input sequence of the automaton and σ' is an output sequence. The automaton A that effectively$_=$enforces $\widehat{P}$ obeys the following properties: a) $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$; b) $\widehat{P}(\sigma')$. In case of invalid input, the automaton A will output some valid sequence (according to the property b) of effective$_=$enforcement) which is not necessary a prefix of the input. Therefore it is not necessarily a Late automaton. □

For example, the automaton from Fig. 2 that effecti-vely$_=$enforces property $\widehat{P}$ for an invalid input `take(1); browse` will output the `warn` action which is not possible for a Late automaton that has to output some prefix of the input.

Let us have a look at the 2nd and 3rd conditions of precise enforcement [2]:

$$2.\ \widehat{P}(\sigma')$$

$$3.\ \widehat{P}(\sigma) \Rightarrow \forall i\ \exists q''.\ (\sigma, q_0) \xrightarrow{\sigma[..i]}_A (\sigma[i+1..], q'')$$

These conditions mean that the automaton will produce an output *in a step-by-step* fashion with the monitored action stream and will output only a valid prefix. As soon as the input sequence becomes illegal, the automaton will no longer produce any output. Therefore, in case of precise enforcement for illegal input, it will output some valid prefix $\sigma' = \sigma[..k]$ such that $\forall i.\, i \leq k.\, \widehat{P}(\sigma[..i]) \wedge \neg\widehat{P}(\sigma[..k+1])$. In case of late precise enforcement for illegal input the output will be some valid prefix $\sigma' = \sigma[..k]$ such that $\forall i.\, i \leq k.\, \widehat{P}(\sigma[..i])$.

Theorem 3. *Edit automata that lately precisely enforce a property $\widehat{P}$ are exactly Late automata, Late automata for $\widehat{P}$ and they effectively$_=$enforce property $\widehat{P}$.*

Proof. Similarly to the definitions of late precise enforcement and effective$_=$enforcement by σ we denote an input sequence of the automaton and σ' is an output sequence.

(*If Direction*). If edit automaton A lately precisely enforces property $\widehat{P}$ then it always outputs a prefix of the input, therefore statement (13) holds and A is a Late automaton. Since the 2nd condition of late precise enforcement states that $\widehat{P}(\sigma')$ then the statement (16) of Late automaton for $\widehat{P}$ holds, therefore A is a Late automaton for $\widehat{P}$. According to Thm. 1, since A lately precisely enforces $\widehat{P}$ it also effectively$_=$enforces $\widehat{P}$.
(*Only-if direction*). If A is a Late automaton, A is Late automaton for $\widehat{P}$ and it effectively$_=$enforces property $\widehat{P}$ then it always outputs some valid prefix of the input (property (13) of Late automaton and property $\widehat{P}(\sigma')$ of effective$_=$enforcement) and in case of valid input it outputs the whole input sequence (property

$\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$ of effective$_=$enforcement). Hence, the 2nd property of late precise enforcement holds: $\widehat{P}(\sigma')$. The 3rd property holds as well because in this case at every step the Late automaton for $\widehat{P}$ outputs some valid prefix and finally it outputs the input sequence because of effective$_=$enforce-ment. Hence A lately precisely enforces $\widehat{P}$. □

Theorem 4. *Edit automata that lately precisely enforces property $\widehat{P}$ are a proper subset of Late automata.*

Proof. First we have to prove that if edit automata A lately precisely enforces property $\widehat{P}$ then A is a Late automaton. Since A lately precisely enforces property $\widehat{P}$ then at every step it outputs valid prefix of the input while Late automaton outputs some prefix. Hence A is a Late automaton.

Next we prove that if A is a Late automaton then it is not necessarily that it lately precisely enforces property $\widehat{P}$. Obviously this statement holds because in case of illegal input a Late automaton can output some invalid prefix while the automaton that lately precisely enforces property $\widehat{P}$ can output only valid prefix. □

For example, the Late automaton presented in Fig. 5 for an input sequence `take(1); browse; pay(1)` will output all actions, while the edit automaton that lately precisely enforces property $\widehat{P}$ cannot output any illegal sequence of actions.

Theorem 5. *All-Or-Nothing automata are a proper subset of Late automata.*

Proof. First we show that if A is All-Or-Nothing automaton then A is Late automaton. Since statement (13) holds for All-Or-Nothing automaton A then A is a Late automaton.

Next we show that if A^* is Late automaton then it is not necessary that A^* is All-Or-Nothing automaton. A^* can output some prefix of the input that can be some prefix of all suppressed actions. In this case A^* is not a All-Or-Nothing automaton because statement 14 does not hold. □

For example, the Late automaton from Fig. 5 for an input sequence `take(1); browse` will output only `take(1)` action. The given Late automaton can not be an All-Or-Nothing automaton because it can output some prefix of the suppressed sequence.

Proposition 6. *All-Or-Nothing automata are not a subset of Late automata for property $\widehat{P}$.*

Proof. We show that if A is All-Or-Nothing automaton then it is not necessary that A is Late automaton for property $\widehat{P}$. An All-Or-Nothing automaton A can output some invalid prefix of the valid input which is not possible for a Late automaton for $\widehat{P}$. □

For example, the All-Or-Nothing automaton shown in Fig. 6 for the input sequence `take(1); pay(1); take(2)` outputs an invalid prefix of this sequence (in this case the whole sequence) while this is not possible for a Late automaton for property $\widehat{P}$.

Proposition 7. *Edit automata that lately precisely enforces property $\widehat{P}$ and All-Or-Nothing automata are not a proper subset of each other.*

Proof. First we have to prove that if edit automaton A lately precisely enforces property $\widehat{P}$ then it is not necessary that A is an All-Or-Nothing automaton.

By σ we denote an input sequence of the automaton and σ' is an output sequence. Automaton A can output some valid prefix of the input which is not necessarily all the suppressed actions, hence A is not All-Or-Nothing automaton.

Next we have to prove that if edit automaton A^* is an All-Or-Nothing automaton then it is not necessary that A^* lately precisely enforces $\widehat{P}$. At some step A^* can output some prefix of the input such that $\neg\widehat{P}(\sigma')$ while automaton that lately precisely enforces $\widehat{P}$ always outputs only valid prefix of the input. □

For example, the automaton given in Fig. 8 that lately precisely enforces property $\widehat{P}$ for an input sequence `take(1); pay(1); take(2)` outputs the sequence `take(1); pay(1)`. The given automaton cannot be an All-Or-Nothing automaton because after `take(2)` action it outputs only the prefix of the suppressed sequence.

On the other hand, for the input sequence `take(1); pay(1); take(2)` the All-Or-Nothing automaton from Fig. 6 will output the input sequence. This automaton cannot be considered as automaton that lately precisely enforces property $\widehat{P}$ because it produces illegal output.

Theorem 6. *All-Or-Nothing automata that lately precisely enforce a property $\widehat{P}$ are exactly Ligatti automata for property $\widehat{P}$.*

Proof. We have to show that All-Or-Nothing automaton A lately precisely enforces a property $\widehat{P}$ if and only if A is a Ligatti automaton for property $\widehat{P}$.

(*If Direction*). If automaton A is All-Or-Nothing automaton then (14) holds. Since A lately precisely enforces property $\widehat{P}$ then according to the Thm. 3:

- A is a Late automaton hence (13) holds;
- A is a Late automaton for $\widehat{P}$ therefore (16) holds;
- A is an All-Or-Nothing automaton that effective-ly$_=$enforces $\widehat{P}$ hence at every step it outputs all suspended sequence or suppress the action (statement (13) holds) and it always outputs the input sequence σ if $\widehat{P}(\sigma)$.

Therefore in case of valid input ($\widehat{P}(\sigma)$) the output is equal to the input, so statement (17) holds as well. We have proved that A obeys all the conditions in the definition of Ligatti automaton for $\widehat{P}$.
(*Only-if direction*). If A is a Ligatti automaton for $\widehat{P}$ then

- It obeys the property (14) hence A is an All-Or-Nothing automaton;
- It obeys the property (13) hence A is a Late automaton;
- It obeys the property (16) hence A is a Late automaton for $\widehat{P}$;
- It obeys the property (17) hence A always outputs input sequence in case of valid input and the longest valid prefix in case of illegal input, hence A effective-ly$_=$enforces $\widehat{P}$.

Therefore since A is Late automaton, A is Late automaton for $\widehat{P}$ and it effectively$_=$enforces $\widehat{P}$ then it lately precisely enforces $\widehat{P}$ according to Thm. 3. □

Now we will clarify which type of edit automaton is constructed following the proof of Thm. 8 in [2] for property $\widehat{P}$ and which type of edit automaton is the one in [2] (Fig. 2).

As the Proposition 3 states, the edit automaton constructed following the proof of Thm. 8 in [2] for property $\widehat{P}$ is a Ligatti automaton for $\widehat{P}$. The edit automaton given in Fig. 2 [2] is an edit automaton that effective-ly$_=$enforces $\widehat{P}$: the 2nd condition of effective$_=$enforce-ment is fulfilled (the automaton always outputs the valid sequence) and the 3rd condition is valid because in case of valid input it always outputs all the sequence. The edit automaton given in Fig. 2 [2] is not a Late automaton because it does not always output some prefix of the input (see examples 7, 11, 12, 15 in Tab. 2)

Therefore we can conclude that both automata from Thm. 8 [2] and from Fig. 2 [2] are edit automata that effectively$_=$enforce property $\widehat{P}$. But when one wants to construct such an automaton and follows the proof of Thm. 8 [2], he obtains a Ligatti automaton for $\widehat{P}$ that lately precisely enforces $\widehat{P}$.

6 From the Policy to the Edit Automata

In previous sections we have presented security property as a predicate $\widehat{P}$ on all possible sequences of executions. Then following the proof of Thm. 8 [2] only infinite state Ligatti Automaton can be constructed.

Proposition 6.24 of [22] states that for any edit automaton A effectively$_=$enforcing property $\widehat{P}$ there exists a Büchi Automaton specifying $\widehat{P}$. The proof of this proposition assumes that the edit automaton is of a particular kind, i.e. a Ligatti Automaton for $\widehat{P}$. Indeed, the authors assume that each state of given edit automaton contains the longest valid prefix σ_A (i.e. the sequence edited by the automaton while reading) and the suffix of the input σ_S that is suppressed by the automaton after reading. Also the construction is made in such a way that all the states of new Büchi Automaton are the same as in the given edit automaton. Every time the edit automaton suppresses an action the next state of the Büchi Automaton is considered to be non-accepting, while when the action is accepted the next state of the Büchi Automaton is considered to be accepting. In this construction an edit automaton can insert only all of those actions that were read before. Therefore this construction can be used only for Ligatti automata for property $\widehat{P}$.

In this paper we reverse the idea of [22] and construct an edit automaton from some automaton that represents our desired security policy. In our model we assume both finite and infinite executions. Since Büchi Automaton accepts only infinite sequences we need another notion of automaton that can represent our security policy.

Definition 9 (Policy automaton). *A* Policy automaton *is a 5-tuple of the form* $\langle \Sigma, Q, q_0, \delta, F \rangle$ *where* Σ *is finite nonempty set of security-relevant program*

actions, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is a labeled partial transition function, and $F \subseteq Q$ is a set of accepting states.

Policy automaton has no output or keep function, it has only accepting states.

Definition 10 (Run of a Policy automaton). *Let $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a policy automaton. A* run *of A on a finite (respectively infinite) sequence of actions $\sigma = \langle a_0, a_1, a_2, \ldots \rangle$ is a sequence of states $q_{|\sigma|} = \langle q_0, q_1, q_2 \ldots \rangle$ such that $q_{i+1} = \delta(q_i, a_i)$. A* finite run is accepting *if the last state of the run is an accepting state. An* infinite run is accepting *if the automaton goes through some accepting states infinitely often.*

The Policy automaton combines the acceptance conditions of Büchi Automata and finite state automata.

Definition 11 (Property represented as Policy Automaton). *Some property $\widehat{P}_A$ is represented as a Policy automaton A if and only if*

$$\forall \sigma \in (\Sigma^* \cup \Sigma^\omega) : \widehat{P}_A(\sigma) \iff A \text{ accepts } \sigma \tag{20}$$

$$\forall \sigma \in (\Sigma^* \cup \Sigma^\omega) : \neg\widehat{P}_A(\sigma) \iff A \text{ does not accept } \sigma \tag{21}$$

Let us now define what kind of properties can be represented as a Policy automaton. In [15] the authors define the class of properties named *Renewal* properties.

Definition 12 (Renewal property). *Property $\widehat{P}$ is renewal if the following holds:*

$$\begin{aligned} \forall \sigma \in \Sigma^\omega : \widehat{P}(\sigma) \iff \\ (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \widehat{P}(\tau)) \end{aligned} \tag{22}$$

According to the Thm. 3.3 [15] a property $\widehat{P}$ can be effectively$_=$enforced by some edit automaton if this property is renewal, $\widehat{P}(\cdot)$ and for all finite sequences σ $\widehat{P}(\sigma)$ is decidable. Therefore we will focus on the renewal properties. The proof of Thm. 3.3 [15] is similar to the proof of Thm. 8 [2] and it is non-constructive because the number of states of the resulting edit automaton is infinite.

Theorem 7. *The set of traces accepted by a Policy automaton is a renewal property.*

Proof. Let us prove the theorem by contradiction. Suppose that there exists a string $\sigma \in \Sigma^\omega$ such that Policy automaton A accepts σ but σ does not satisfy equation (22).

Then there exists a sequence σ', $\sigma' \preceq \sigma$ such that $\forall \tau. \tau \preceq \sigma. \sigma' \preceq \tau. \neg\widehat{P}(\tau)$. In this case there exists a run $s = \langle s_0, s_1, \ldots, s_d, \ldots \rangle$ for a sequence of actions $\sigma = \langle a_1, \ldots, a_d, \ldots \rangle$ such that s_d is *not* an accepting state. Since σ is accepted by A there must be a successor state of s_d that is accepting (otherwise s would have only finitely many accepting states) i.e. a subsequence of

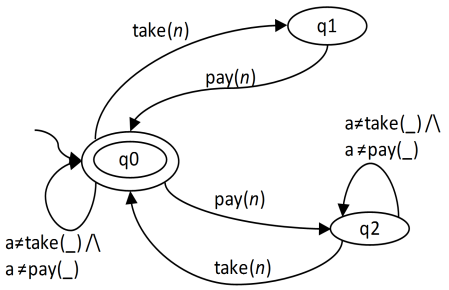

Fig. 10. Policy automaton representation of market policy (Ex. 1)

$s = \langle s_0, s_1, \ldots, s_d, \ldots, s_l \rangle$ such that at least s_l is accepting then the corresponding sequence of actions $\tau_l = \langle a_1, \ldots, a_d, \ldots, a_l \rangle$ is such that $\sigma' \preceq \tau_l \preceq \sigma \wedge \widehat{P}(\tau_l)$ which is a contradiction. □

The converse of the theorem is obviously not true because the language $\{\sigma \notin \Sigma^\omega | \, \widehat{P}(\sigma)\}$ can be more expressive than a language acceptable by a Büchi Automaton. For instance the language $\{((a+b)^* a^n b^n)^\omega | \, n \in N\}$ cannot be generated by a Büchi Automaton.

In Fig. 10 we present a Policy automaton for the market policy from Ex. 1. According to the Ex. 1 in English, the automaton should accept all the sequences when `take-pay` transaction is finalized. Otherwise if after `take(n)` action there is some action different from `pay(n)` then the policy is violated and the automaton halts. If after `pay(n)` action there are some other actions different from `pay` and `take` then the automaton simply waits for the `take(n)` action. In case of `take(m)` action when $m \neq n$ the automaton halts. In this way we give our own interpretation to the given example.

For given renewal property $\widehat{P}$ represented as Policy automaton we tried to present a construction of Ligatti automaton that is the inverse of of Talhi et al. [22] construction from Proposition 6.24.

Intuitively, resulting Ligatti automaton should have the same number of states and transitions as given Policy automaton. At every transition that goes to non-accepting state Ligatti automaton should suppress the input action and add it to some suppressed sequence σ. When the next state of Policy automaton is accepting Ligatti automaton should output the whole suppressed sequence.

If given property $\widehat{P}$ accepts at least one infinite sequence then the policy automaton representing it has at least one cycle over accepting state. Then when we construct Ligatti Automaton for Policy automaton from Fig. 10 instead of state $q2$ we will have infinite number of states, each of them will contain some different sequence of suppressed actions: `pay(n)`, `pay(n)`; a, `pay(n)`; a; a etc.

Hence the constructed Ligatti automaton should have some keep function γ_k that defines all the suppressed actions. In the construction algorithm from the proof of Thm. 8 [2] (see Fig. 3) every state itself included suppressed sequence, hence there were infinite number of states. When one wants to construct Ligatti

Algorithm 1 LigattiAutomaton(A) Function

Input: PolicyAutomaton $A = \langle \Sigma, Q, q_0, \delta, F \rangle$;
Output: LigattiAutomaton $A^E = \langle \Sigma, Q^E, q_0^E, \delta^E \rangle$ that effectively$_=$enforces the policy represented by A;

1: $q_0^E := q_0$;
2: $Q^E := Q \cup \{q_\perp\}$
3: **for all** $(a \in \Sigma)$ **do**
4: **for all** $(q \in Q)$ **do**
5: **if** $\delta(q, a)$ is defined **then**
6: $\delta^E(q, a) := \delta(q, a)$;
7: **if** $\delta(q^E, a) \in F$ **then**
8: $\gamma_o(q^E, a, \sigma) := \sigma; a$;
9: $\gamma_k(q^E, a, \sigma) := \cdot$;
10: **else**
11: $\gamma_o(q^E, a, \sigma) := \cdot$;
12: $\gamma_k(q^E, a, \sigma) := \sigma; a$;
13: **else**
14: $\delta^E(q, a) := q_\perp$;
15: $\gamma_o(q^E, a, \sigma) := \cdot$;
16: $\gamma_k(q^E, a, \sigma) := \cdot$;
17: $\delta^E(q_\perp, a) := q_\perp$;

automaton the keep function γ_k should define all the actions that are kept while input sequence is invalid. Then as soon as some next action makes the whole sequence valid (i.e. accepted by Policy automaton), the output function γ_o should output all the suppressed actions and the result of the γ_k function should be an empty sequence.

The constructive algorithm of Ligatti automaton for property expressed by the Policy automaton A is shown in Alg. 1. Following this algorithm we construct a Ligatti automaton shown in Fig. 11

If we compare Ligatti automaton from Fig. 3 and Ligatti automaton from Fig. 11 we will see that their output is identical for the same input. The difference is that the automaton built by a proof of Thm. 8 [2] has infinite number of states while we provide an extended finite representation of Ligatti automaton for renewal property represented as Policy automaton.

Practically the σ keep sequence can be easily implemented by a queue. The Ligatti automaton has some queue that keeps all the suspended actions (this notion is similar to a very restricted form of *Queue Automaton* [3]). In our particular case the γ_o function outputs all actions in the queue (or not at all) and γ_k function only enqueues elements in the queue (when there is no output) or reset the queue to the empty one.

Theorem 8. *Any security policy, represented as a Policy automaton A can be effectively$_=$enforced by some Ligatti automaton A^E.*

Proof. We construct a Ligatti automaton A^E following the Alg. 1. This automaton has γ_o and γ_k functions that define the output for the transitions

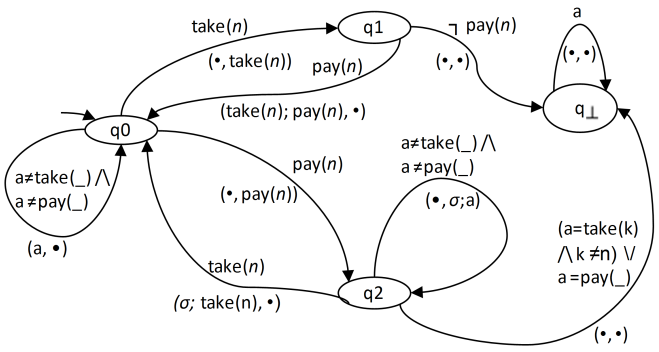

Fig. 11. Finite representation of Ligatti automaton constructed for a Policy automaton (Fig. 10)

and sequence that will be kept after commiting the transition. Automaton A^E effectively$_=$enforces the security policy represented as Policy automaton A^P because all conditions of effective$_=$enforcement are satisfied. Indeed, the 2nd condition is true because the automaton A^E outputs some sequence of actions σ only when the reached state is accepting (this statement is equal to $\widehat{P}(\sigma)$). The 3rd condition of effective$_=$enforcement is satisfied as well because when the input sequence is valid it means that the current state is an accepting state, hence we will output the whole input sequence. □

Corollary 1. *If the Policy automaton is finitely represented then the Ligatti automaton is also finitely represented.*

Proof. For a given Policy automaton we construct a Ligatti automaton following the Alg. 1. According to the construction algorithm, the resulting automaton will have a finite representation. □

7 Related work and Conclusions

Schneider [20] was the first to introduce the notion of enforceable security policies. The follow-up work by Hamlen et al. [8] fixed a number of errors and characterized more precisely the notion of policies enforceable by execution monitors as a subset of safety properties. They also analyzed the properties that can be enforced by static analysis and program rewriting. This taxonomy leads to a more accurate characterization of enforceable security policies. Ligatti, Bauer, and Walker [2] have introduced edit automata; a more detailed framework for reasoning about execution monitoring mechanisms. As we already said, in Schneider's view execution monitors are just sequence recognizers while Ligatti et al. view execution monitors as sequence transformers. Having the power of modifying program actions at run time, edit automata are provably more powerful than security automata [14].

Fong [6] provided a fine-grained, information-based characterization of enforceable policies. In order to represent constraints on information available to execution monitors, he used abstraction functions over sequences of monitored programs and defined a lattice on the space of all congruence relations over action sequences aimed at comparing classes of EM-enforceable security policies. Still his policies are limited to safety properties over finite executions.

Martinelli and Matteucci [16] have shown how to synthesize program controllers that monitor behavior of the untrusted components of the system. Given the system and a security policy represented as a μ-calculus formula the user can choose the controller operator (truncation, suppression, insertion or edit automata). Then he can generate a program controller that will restrict the behavior of the system to those specified by the formula.

When a security policy is represented by a predicate $\widehat{P}$ over set of finite executions we can conclude that both automata from Thm. 8 [2] and from Fig. 2 [2] are edit automata that effectively$_{=}$enforce property $\widehat{P}$. If one wants to construct such an automaton and follows the proof of Thm. 8 [2], he obtains a Ligatti automaton for $\widehat{P}$ that lately precisely enforces $\widehat{P}$. A problem that is present in the construction of Thm. 8 is that it assumes an oracle that can tell for each sequence σ whether $\widehat{P}(\sigma)$ holds or not. A security policy in Thm. 8 [2] is a predicate $\widehat{P}$ on all possible finite sequences of executions, but in this case the edit automaton which effectively enforces this policy is only of theoretical interest: following the proof of Thm. 8 only infinite states automata can be constructed.

In summary, we have shown that the difference between the running example from [2] and the edit automata that are constructed according Thm. 8 in the very same paper is due to a deeper theoretical difference. In order to understand this difference we have introduced better classification of edit automata introducing the notion of *Late Automata*. The particular automata that are actually constructed according Thm. 8 from [2] are a particular form of late automata that have an all-or-nothing behavior and that we named Ligatti's automata after their inventor.

Hence, the construction from Talhi et al. [22] only applies to Ligatti's automata. Given a (infinite state) Ligatti automaton they can extract the Büchi automaton that represent the policy effectively enforced by the Ligatti automaton. What happens if the automaton is not a Ligatti automaton? For example the automaton from Fig. 2? Proposition 6.24 [22] simply does not apply. It needs to be shown whether given a general edit automaton one can construct a Büchi automaton so that the latter represents the policy that is effectively enforced by the former. We leave this question open for future investigation.

What remains to be done? Our results shows that the edit automaton that you can actually write (e.g. by using Polymer) does not necessarily correspond to the theoretical construction that provably guarantees that your automaton enforces your policy.

So we fully re-open the most intriguing question that the stream of papers on execution monitors seemed to have closed:

Challenge 1 *You have written your security enforcement mechanism (aka your edit automata); how do you know that it really enforces the security policy you specified?*

Our constructive proof of Thm. 8 is only a first step to address this research challenge. Given a policy specification expressed as a Policy automaton or Büchi automaton (as used in Security-by-Contract [4, 17, 18]) we constructed an extended finite state automaton that effecti-vely$_{=}$enforced it.

There is however a much broader issue we would like to raise. Essentially all papers on security monitors cited in this article (and this paper itself) only provide a security judgment over a trace (i.e. a predicate over trace) that considers the trace as a whole. Hence we are not able to define an incremental notion of security that tells how to fix a bad trace. The Ligatti automaton will output only the longest valid prefix, the only available theoretical fix is amputation. So we open another question:

Challenge 2 *If your enforcement mechanism really enforces your security policy, how exactly it does the enforcement? Does it fix the bad sequences in the way you want?*

References

1. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 305–314. ACM Press, 2005.
2. L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.
3. A. Cherubini, C. Citrini, S. Crespi Reghizzi, and D. Mandrioli. Qrt fifo automata, breadth-first grammars and their relations. *Theoretical Computer Science*, 85(1):171–203, 1991.
4. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proceedings of the 4st European PKI Workshop: Theory and Practice*. Springer-Verlag, 2007.
5. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical report 2003-1916, Department of Computer Science, Cornell University, 2003.
6. P.W.L. Fong. Access control by tracking shallow execution history. *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, May 2004.
7. L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.
9. J. Hartmanis. *Algebraic structure theory of sequential machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1966.
10. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technol. Transfer*, 2004.

11. K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *Proceedings of the 12th ACM Conference on Communications and Computer Security*, 2005.
12. B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
13. J. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.
14. J. Ligatti, L. Bauer, , and D. Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security*, pages 355–373, 2005.
15. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 2008. Available at `http://www.cs.princeton.edu/~dpw/papers/run-time-enforcement.pdf`.
16. F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. In *Proceedings of the Second International Workshop on Security and Trust Management*. Lecture Notes in Computer Science, 2006.
17. F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proceedings of The 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.
18. F. Massacci and I. S. R. Siahaan. Simulating midlet's security claims with automata modulo theory. In *Proceedings of the 2008 workshop on Programming Language and analysis for security*, pages 1–9. ACM, 2008.
19. Bill Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/`, May 2007.
20. F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
21. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 15–28. ACM Press, 2003.
22. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2-4):158–184, 2007.