PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

A theory of constructive and predictable runtime enforcement mechanisms

Nataliia Bielova

November 2011

Abstract

Nowadays owners and users of software systems want their executions to be reliable and secure. Runtime enforcement is a common mechanism for ensuring that system or program executions adhere to constraints specified by a security policy. It is based on two properties: the enforcement mechanism should leave legal executions without changes (transparency) and make sure that illegal executions are amended (soundness).

From the theory side, the literature proposes the precise characterization of legal executions that represent a security policy and thus is enforced by mechanisms like security automata or edit automata. Unfortunately, transparency and soundness do not distinguish what happens when an execution is actually illegal (the practical case). They only tell that the outcome of an enforcement mechanism should be "legal", but not how far the illegal execution should be changed.

In this thesis we address the gap between the theory of runtime enforcement and the practical case. First, we explore a set of policies that represent legal executions in terms of repeated legal iterations and propose a constructive enforcement mechanism that can deal with illegal executions by eliminating illegal iterations. Second, we introduce a new notion of predictability, that puts a restriction on the way illegal executions are modified by an enforcement mechanism. Third, we propose an automatic construction of enforcement mechanisms that is able to tolerate some insignificant errors of the user and we prove it to have a sufficient degree of predictability.

The main case study of this thesis is a business process from a medical organization. A number of discussions with the partners from this organization shows the validity of the approaches described in this thesis in practical cases.

Keywords
security policy enforcement; edit automata; runtime monitors; error-toleration

Contents

List of Tables

List of Figures

Acknowledgements

While doing my PhD I have learned a lot and I am truly grateful to many people who were there for me and helped me to discover how to do research and supported me in good and bad times.

First of all, I would like to thank my advisor Fabio Massacci for a guidance in the beginning of my life as a researcher and for a lot of useful advices through all these years. I would also like to thank my teacher of mathematics, Svetlana Vitalievna Voronina. If she did not show me the beauty of mathematics, I would never become a researcher. The great influence on my scientific life was made by Aleksandr Leonidovich Khizha, who inspired me by his excellent teaching of Program Verification and showed me how a man can be excited about his scientific work. Another thank you goes to Ida Siahaan, Katsiaryna Naliuka and Nicola Dragoni, who have helped me a lot during my first steps in the PhD life. I would also like to thank Alexander Pretschner for several motivating discussions that we had in the last several years.

I would like to thank several people of DistriNet research group at Katholieke Universiteit Leuven, in particular a special thank you to Frank Piessens and Dominique Devriese for a delightful and very inspiring work that we have done together during my internship.

Finally, I would like to thank my friends and colleagues that have been there for me while I was growing as a researcher. In particular, a special thank you goes to Stephan Neuhaus for a great help and a lot of motivating discussions; another special thank you goes to Gabriela Gheorghe for a great support not only as a colleague, but also as a great friend; I would like to thank much more people from the ICT Doctorate School and form a Russian community in Trento but unfortunately cannot mention here all the names.

A particular thank you goes to Tania for her friendship and support, especially in the last year of my PhD. I would also like to thank my sister Lena for not letting me down in the hard times and cheering me up even more in the good times. I would like to thank my dear Marco for his amazing patience, his true love, his restless care and his great support even when he is not next to me, and I would like to thank my wonderful parents for their endless care and love.

Chapter 1

Introduction

1.1 The Importance of Execution Monitoring

Nowadays owners of business processes want their executions to be secure and compliant with different regulations, including privacy; mobile device users want their private information to stay on their devices; the users of web applications want to be protected from the hackers' attacks. Unfortunately the features of those applications are at odds with the current security models.

The first problem is that the developers of largely used applications such as mobile games or social network applications claim to be compliant with the privacy policies of their company, but they are not obliged to be compliant with security policies of the end users. However, the end users start having more and more security requirements.

The second problem is that once an application is installed, there is no general control over what the application is doing at its runtime. It can potentially collect some amount of private information and send it to the remote server. For example, BGR (Boy Genious Report weblog, a pioneer in breaking news within the mobile gadget sector) have revealed the fact that the Facebook application once it is installed on a smartphone, synchronizes all phone numbers from the smartphone contact list and stores them on Facebook server [32].

Model carrying code [60] or Security-by-Contract [11] claims that there is a solution to this problem. These approaches propose to equip the code with security claims that are later matched against the platform security and privacy policies. This matching checking is done when the program is going to be deployed on the mobile device. However, there are some properties of programs that cannot be checked at deployment time.

To overcome these drawbacks, a number of authors have proposed to enforce the compliance of applications with security policies by execution monitoring. The idea is to monitor the execution of the program at runtime and control its compliance to the security policies. Traditional *security automata* [59] were essentially sequence recognizers that stopped the execution as soon as an illegal sequence of actions (not compliant with the policy) was on the eve of being performed. They were proven to enforce *safety* properties.

In the following years a number of refinements have been proposed, for example Hamlen's work on rewriting [45] and Ligatti et al. works on *edit automata* [7, 51]. The latter work proposes a model of mechanisms that are not only recognizing the correct behavior of applications, but are also capable of transforming their behavior. This power gives edit automata the capability of enforcing more than safety properties and it was proven that they are able to enforce a richer class of properties, called *renewal* properties.

The fact that an edit automaton provably and effectively enforces a given security policy was formalized by two notions: soundness (all transformed executions comply with the policy) and transparency (the semantics of compliant executions should not be changed).

1.2 Problem Statement

Transformation of non-compliant executions We started our research by trying to formally show "as an exercise" that the running example of edit automaton from [7] provably enforces a given security policy by applying the effective enforcement theorem (Theorem 8) from the very same paper. Unfortunately, we failed.

As a result of this failure, we decided to make a deeper investigation and discovered that the impossibility of reconciling the running example with the theorem on the very same paper is a consequence of a gap between the edit automata that one can possibly write and the edit automata that can be constructed by existing techniques (Theorem 8 from [7], Theorem 3.3 from [51] or by Talhi et al. [61]).

Issue 1 *Two edit automata are effectively enforcing the same security property. Can we formally capture what makes them different?*

Automatic construction of enforcement mechanism When an enforcement mechanism is specified manually (as in many state-of-the-art papers and as in the example of edit automaton from [7]), a number of mistakes can be made, since one should precisely define how the illegal executions are transformed into the legal ones and the notions of soundness and transparency do not help to define this transformations.

Issue 2 *There is no generic algorithm to construct an enforcement mechanism from the given security property, such that this mechanism enforces the property in question in a desirable way.*

Notions of soundness and transparency are not sufficient We have found out that different mechanisms provide different enforcement for the same security policy, even though these mechanisms are sound and transparent. In the current theory of runtime enforcement there is no notion that gives at least some information on how the non-compliant execution sequences should be modified by an enforcement mechanism.

Issue 3 *Soundness and transparency only describe what happens to legal executions. A new notion should be able to discriminate execution transformers on the basis of what happens to the illegal executions.*

Toleration of user insignificant errors In practice a security policy is a description of the compliant behavior of the given system. When applied to business processes, it describes the desirable behavior of a business process, for example, in compliance with applicable rules and regulations. Runtime enforcement mechanisms in these systems would observe the actual executions of the processes and control their compliance.

In this particular setting, a policy representing the compliant behavior of the system can be seen as a protocol that the system should follow. Enforcing such a policy means insuring that the system behaves according to the predefined protocol. Since this thesis focuses on building runtime enforcement mechanisms in this thesis, we will use notions from the runtime enforcement theory; hence the notion of "policy" will be used also to describe the "official protocol" of the system behavior.

There are two main requirements to the enforcement mechanisms: 1) the resulting processes must be compliant with the desired behavior ("policy"); 2) the end users should not be disturbed in their primary mission (for example doctors and nurses in the hospital should deliver drugs to the patients) even if an insignificant deviation from the desired process happened. These possible insignificant deviations are difficult to write, to check and to communicate to the users. Hence we think that the runtime enforcement mechanism should be able to tolerate such deviations and manage them as automatically as possible.

Issue 4 *There is no generic construction of runtime enforcement mechanism that can tolerate insignificant deviations from the specified policy.*

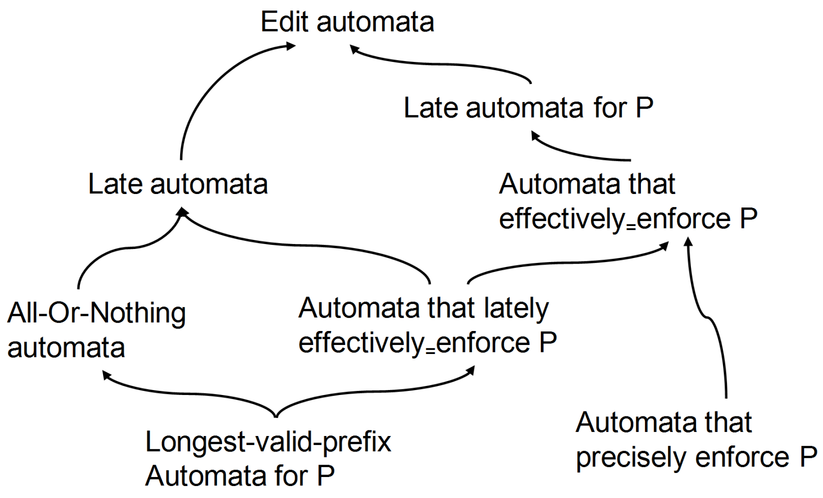

Edit automata
Late automata for P
Late automata
Automata that effectively$_=$enforce P
All-Or-Nothing automata
Automata that lately effectively$_=$enforce P
Longest-valid-prefix Automata for P
Automata that precisely enforce P

Figure 1.1: Hierarchy of edit automata

1.3 Contributions

The work in this thesis focuses on runtime enforcement mechanisms, their models and properties. The investigation started with the gap found in the runtime enforcement theory and a number of solutions have been proposed. The thesis is structured in a way to reflect several main contributions.

1.3.1 Classification of Enforcement Mechanisms

We start with presenting the first result of our investigation. We address Issue 1 by analyzing the examples from [7]. We have built a hierarchy of several kinds of edit automata and found a relation between them.

Figure 1.1 shows the relation between the classes of edit automata that we have established. *All-Or-Nothing automata* output the whole execution sequence or suppress the next incoming action. The notion of effective$_=$enforcement comprises soundness and transparency. *Late automata* are a particular kind of edit automata that always output some prefix of the input. *Longest-valid-prefix automata for P* are automata constructed according to the proof of Theorem 8 of [7]. We give more details in the main body of the thesis. This contribution has led to the following workshop and follow-up international journal publication:

- N. Bielova and F. Massacci. Do you really mean what you actually enforced? In *Proceedings of the 5th International Workshop on Formal Aspects in Security and*

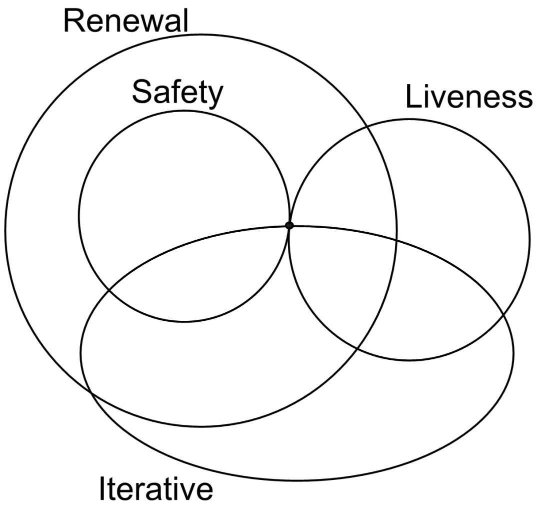


Figure 1.2: Relationships between security properties

Trust, volume 5491 of *Lecture Notes in Computer Science*, pages 287-301. Springer-Verlag Heidelberg, 2008. [12]

- N. Bielova and F. Massacci. Do you really mean what you actually enforced? *International Journal of Information Security*, pages 239-254, 2011. DOI: 10.1007/s10207-011-0137-2. [14]

1.3.2 Automatic Construction of Enforcement Mechanisms and Iterative Properties

We propose an algorithm for the automatic construction of enforcement mechanisms given a security policy represented as an automaton that combines the acceptance conditions of a finite-state automaton and a Büchi automaton. The construction is given for two types of mechanisms. The first one always outputs the longest valid prefix of a tentative execution sequence. We call it *Longest-valid-prefix automaton.*

The second mechanism, called *Iterative Suppression automaton*, is a novel runtime enforcement technique. It outputs the biggest subpart of the tentative execution sequence, assuming that a legal execution is usually a repeating concatenation of some "default" sequences, that we call *iterations*. This mechanism is also proven to be as sound and transparent as any other existing mechanism. Iterative Suppression automata make it possible to distinguish between different types of enforcement for the same security property. When proposing the construction, we use our assumption that compliant executions

Table 1.1: Properties of enforcement mechanism.

Name	Pre-Condition	Post-Condition
Soundness		for every trace the output is always some valid trace
Transparency	if input is a valid trace	output is *the same* valid trace
Predictability	if input is within δ from a valid trace	output is within ε from *the same* valid trace

consist of iterations and prove that Iterative Suppression automata are capable of enforcing a new class of security properties, called *iterative properties*. By this contribution we propose two solutions to Issue 2.

We have established the relations between iterative properties and safety, liveness and renewal properties. We show our results in Figure 1.2 and discuss it in more details in the thesis. This work has led to the following conference and journal publications:

- N. Bielova, F. Massacci, and Andrea Micheletti. Towards practical enforcement theories. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838 of *Lecture Notes in Computer Science*, pages 239-254. Springer-Verlag Heidelberg, 2009. [17]

- N. Bielova and F. Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*, 2011. To appear. [15]

1.3.3 Predictability

We address Issue 3 and argue that what distinguishes enforcement mechanisms is not what happens when the executions comply with the policy (because nothing should happen according to transparency), but what happens when they are not compliant. We propose a notion of *predictability* that puts a restriction on enforcement mechanisms so that they should not change the non-compliant executions in an arbitrary way.

The idea behind this new notion is defined in the spirit of continuity in real-functions. In Table 1.1 we show a comparison of this new notion with soundness and transparency. This work has led to the following publication:

- N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, volume 6542 of *Lecture Notes in Computer Science*, pages 73-86. Springer-Verlag, 2011. [16]

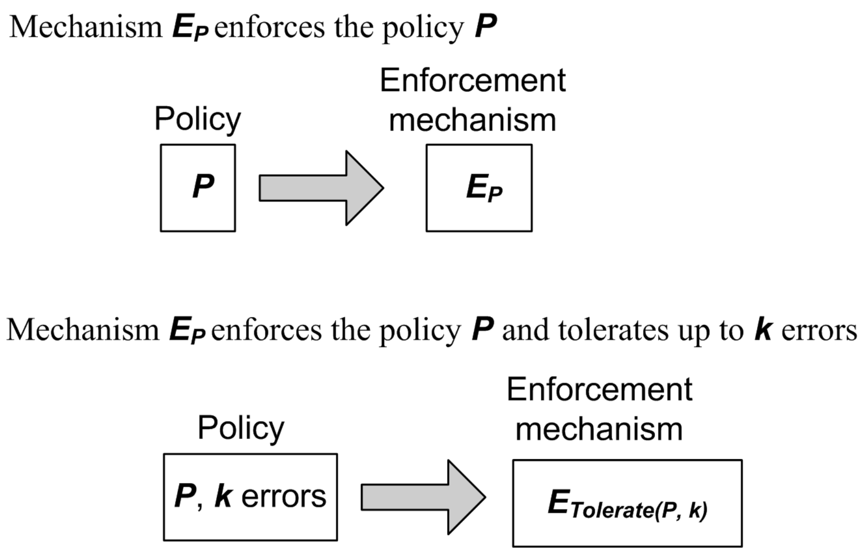

Mechanism E_P enforces the policy P
Policy
Enforcement mechanism
P
E_P
Mechanism E_P enforces the policy P and tolerates up to k errors
Policy
Enforcement mechanism
P, k errors
$E_{Tolerate(P, k)}$

Figure 1.3: Original approach and a novel approach to construction of enforcement mechanisms.

1.3.4 Error-toleration

We build upon the works of automatic policy generation and runtime enforcement to propose a semi-automatic way to generate enforcement mechanisms that can tolerate up to k errors, given a "default" workflow and a specification of a simple list of errors and possibly their corrections. In this way we address Issue 4.

We have proposed an algorithm to construct a particular kind of edit automata that perform this transformation in a well-defined way. This mechanism extends the classical default-deny policy, considering the type and number of deviations from the policy that the edit automaton can allow or amend. Differently from original approach, where the runtime enforcer is constructed from the given security policy, we propose a novel approach to build a runtime enforces that tolerates up to k deviations. We present the difference between the approaches in Figure 1.3. This work has led to the following international conference publication:

- N. Bielova and F. Massacci. Computer-aided generation of enforcement mechanisms for error-tolerant policies. In *Proceedings of the International Symposium on Policies for Distributed Systems and Networks (POLICY'11)*, IEEE Computer Society Press, 2011. [13]

1.3.5 Other Contributions

Besides the contributions described in this thesis, the author of the thesis contributed to several other research results during her PhD studies:

Implementation and evaluation of Security-By-Contract In the Security-by-Contract framework the code of the third-party application carries with itself the *contract* that describes the security relevant behavior of the application; the *security policy* of the platform is deployed on the mobile device.

We have implemented the contract-policy matching, which is one of the key steps of the overall Security-by-contract framework. Contracts and policies are specified in ConSpec [1]. The first description of the implementation appeared in [9], with more results on testing in [18]. The major result is published in [11].

Reactive non-interference for a Browser Model We have worked on enforcing non-interference in the model of the web browser called Featherweight Firefox [19]. Taking the notion of *reactive non-interference*[20] specifically designed for the models of the web browsers, we developed a new enforcement mechanism [10] based on the idea of secure multi-execution [30].

1.4 Structure of the Thesis

The thesis consists of the following chapters:

Chapter 2 surveys the related work in theoretical aspects of runtime monitoring and enforcement of security policies.

Chapter 3 introduces the gap found in the theory of runtime enforcement between the security policies and their corresponding enforcement mechanisms. The main issues that will be solved in this thesis are introduced. While defining the problem, we present several running examples that will be used in the other Chapters of this thesis.

Chapter 4 presents the relation among different types of security policies and establishes a new classification of enforcement mechanisms. This classification is useful to define a relation between two enforcement mechanisms that enforce the same security policy in different ways.

Chapter 5 provides two main constructions of enforcement mechanisms given a security policy. Longest-valid-prefix automaton outputs the longest valid prefix of the given system execution and Iterative Suppression automaton suppresses the illegal parts of the execution. We also discuss how the enforcement is done for the running examples defined in Chapter 3.

Chapter 6 introduces a new notion of predictability that defines how the enforcement mechanism should behave when the system executions do not comply with the given policy. This Chapter also presents a construction of a new runtime enforcement mechanism that can tolerate a number of insignificant errors.

Chapter 7 presents a concluding discussion.

Chapter 2

State of the Art

This chapter presents the state of the art techniques for runtime enforcement of security policies. We describe here two main types of enforcement mechanisms: first type are only able to recognize whether the system execution satisfies the desired policy (sequence recognizers) and second type are also able to transform the system executions at runtime (sequence transformers).

2.1 Systems, Policies and Properties

At the highest level of abstraction we will model the system execution with *traces*, where a trace is a sequence of actions emitted by the execution. Usually in the state of the art literature the executing system being monitored is called the *target*. The targets can be objects, processes, subsystems or entire systems. The actions being monitored are execution steps of the target that may range from low level operations (such as memory access actions) to high level operations (such as method calls).

The *security policy* defines expected (or correct according to some security concerns) behavior of the target and is usually represented in terms of set of possible execution traces of the target. In order to ensure that the target behaves according to the security policy, the *runtime enforcement* mechanism is deployed. It monitors the target execution and enforces the security policy by terminating or modifying the original executions when they violate the security policy. Originally, runtime monitor was named *Execution Monitor (EM)* by Schneider [59].

A security policy defines an expected behavior of the target. As it was originally proposed by Schneider [59], security policy is specified as a predicate P on sets of (finite- or infinite-length) execution sequences. A target satisfies security policy P if and only if the set of all its execution sequences satisfies this policy.

As it is defined by Schneider [59] and refined by Ligatti et al [51], *security properties* are a strict subset of security policies and they are distinguished as follows. A security policy P is a property if there exists a characteristic predicate $\widehat{P}$ over individual execution sequences. Then whenever a security policy P holds for a set of execution sequences, its corresponding predicate $\widehat{P}$ holds for every sequence in that set.

Not every security policy is a security property. Since the property is only defined in terms of single execution sequences, it cannot specify the relations between different sequences. For example, it is a known fact that an information flow policy is not a security property since it is defined in terms of sets of target execution sequences and relations between them.

In the state of the art there are several classes of security properties. The property describing behavior such as "nothing bad ever happens" is called *safety* property [2, 49]. Safety means that whenever an execution sequence is valid, all its prefixes are valid as well. More intuitively, safety means that when a violation happens (meaning a sequence is invalid), there is no way to remediate it (there is no suffix that can make it valid again).

Additionally to safety, there is another class of properties called *liveness* [2]. These properties describe the behavior such as "something good eventually happens during any execution". In other words, this means that any finite execution sequence has a valid continuation.

Not all the properties are pure safety or pure liveness. Some properties may allow an execution sequence to alternate between satisfying and violating the property. As it was shown by Alpern and Schneider [3], all the properties can be described as a combination of safety and liveness.

When analyzing the enforcement capabilities of different mechanisms, Ligatti et al. [51] defined a new class of properties, called *infinite renewal properties.* According to this property, every valid infinite-length sequence has infinitely many valid prefixes, and every invalid infinite-length sequence has only a finite number of valid prefixes. For example, let τ range over finite sequences of legal transactions. Then a transaction policy accepting τ^∞ is a renewal property: any valid infinite-length sequence satisfying the transaction policy τ^∞ has infinitely many valid prefixes of the form τ^*. Any invalid infinite-length sequence that does not satisfy the transaction policy is of the form $\tau^*; \sigma$, where σ is not a prefix of τ, which means that this sequence has only a finite number of valid prefixes.

Given this classification of security properties, different authors started to propose enforcement mechanisms that are capable of enforcing particular types of properties.

2.1.1 Hyperproperties

Later Clarkson and Schneider [28] introduced *hyperproperties* that are sets of properties, or sets of sets of execution traces. Hence, hyperpoperties are able to describe security policies that are not properties, such as information-flow policies. The authors also propose a notion of hypersafety and hyperliveness, that are also hyperproperties and are specified in a similar way as safety and liveness, with the only difference that instead of traces they use sets of traces. Interestingly, the authors generalize the result of Alpern and Schneider [2] that every security property is the intersection of safety property and liveness property. Clarckson and Schnieder prove that every hyperproperty is the intersection of hypersafety and hyperliveness.

Another interesting result from [28] is that noninterference, as defined by Goguen and Meseguer [41], can be written as a particular type of hypersafety called *k-safety hyperproperty.* The idea behind k-safety is that the "bad thing" defining that the continuations of system executions will not satisfy this hyperproperty, never involves more than k traces. Since Goguen and Meseguer's definition of noninterference compares two traces, it is a 2-safety hyperproperty. In [28] the authors also suggest a relatively complete verification technique for k-safety hyperproperties.

The hyperproperties, especially hypersafety and k-safety hyperproperty, are very interesting for investigation of their enforcement techniques, however this thesis is covers the area of enforcement for security properties that can be specified with a single predicate on executions traces.

2.2 Runtime Enforcement Mechanisms as Sequence Recognizers

The first idea of modeling an enforcement mechanism for security properties belongs to Schneider who proposed to think about runtime monitors as sequence recognizers.

2.2.1 Security Automata

Schneider introduced the notion of enforceable security policies and Execution Monitoring (EM) in 2000 [59]. He was the first proposing the notion of security properties and trying to analyze security policies enforceable by execution monitoring. In his paper, Schneider fairly claims that a security policy must be a property in order for a policy to have an enforcement mechanism is EM. This statement is correct since a runtime monitor makes decision about a single execution at runtime and is incapable of comparing different executions between themselves. Having that in mind, all the follow-up work that has been done in the area discusses only runtime enforcement mechanisms for security properties.

Also, the author tried to restrict the obtained result stating that "If the set of executions for a security policy P is not a safety property, then an enforcement mechanism from EM does not exist for P." This statement, however, is not true. The problem with this statement is that the terms used in it do not correspond to the notions that the author obviously had in mind. By Execution Monitors the author means sequence recognizers that can only conclude whether the execution satisfies the property or not. These mechanisms have the power of accepting a single action at runtime or terminate the whole execution. Having that in mind, the statement becomes true, hence one could rewrite it as "If the set of executions for a security policy P is not a safety property, then an enforcement mechanism modeled as sequence recognizer does not exist for P."

Schneider proposed a first runtime enforcement mechanism modeled as sequence recognizer, called *Security Automata (SA)*. It monitors the execution of the target and halts it as soon as it violates the property. In this way the proposed mechanism naturally enforces safety properties.

2.2.2 Shallow History Automata

Fong [38] provided a new approach to classify enforceable security properties. First, he proposed to investigate an open question raised by Bauer et. al in [6]: whether it is possible to further classify the space of EM-enforceable policies by constraining the capabilities of the execution monitor.

Fong successfully found the way to prove that this is possible by classifying the space of EM-enforceable policies according to the information consumed by an execution monitor. He proposed a new mechanism called *Shallow-History automata (SHA)* that keeps as a history the access control events occurred in the past, and does not contain any information about the order of their arrival. By doing so, Fong proved that SHA is strictly less expressive than SA.

To show the practicality of his mechanism, Fong showed how some well known security policies such as Chinese wall policy [21], low-water-mark policy [8] or one-out-of-k authorization policy [31] can be enforced by SHA. Later he proved that the class of policies enforceable by SHA is less expressive than the general class of EM-enforceable policies. Fong proposed to use the generalization of his technique to define a lattice of security policy classes, in which member classes are ordered by the amount of information that must be tracked by an execution monitor. However, this classification applies only to safety properties.

2.2.3 Computational power of Execution Recognizers

Viswanathan, Kim et al. [48, 62] decided to investigate the computational power of runtime monitoring mechanisms, in particular they limited their work on execution recognizers enforcing safety properties in finite time. They introduced computability constraints and showed that the monitors recognizing invalid executions enforce the set of computable safety properties.

Hamlen et al. [45] revised the classifications given by Schneider [59] taking into account work done by Viswanathan and Kim et al. In order to analyze the computational power of runtime enforcement mechanisms, Hamlen et al. first give a set of assumptions that are relevant for all the preceding and subsequent works on runtime monitors. In particular, since runtime enforcers should present security policy violations before they occur, they always must have a power of predicting target's behavior on any given input. For example, a security automaton must be able to look ahead at least one computational step in order to intercept bad events before they occur.

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.

2.3 Runtime Enforcement Mechanisms as Sequence Transformers

2.3.1 Edit Automata

Later Ligatti, Bauer, and Walker [7, 51] have introduced *edit automata*, a new mechanism that is capable of enforcing a class of security properties strictly bigger than a class of safety properties. The authors could achieve such a result because, differently from Schneider that considers runtime enforcement mechanisms as sequence recognizers, they propose to view them as sequence transformers.

Remark 2.3.1 *Notice that before appearance of runtime enforcement mechanisms as execution transformers, runtime enforcement was often done by runtime monitoring. In* *other words, monitors were considered "enforcers" as they were able to observe the system executions and halt them as soon as they violate the given security properties. With the appearance of sequence transformer techniques, runtime monitoring and runtime enforcement are two distinctive notions.*

Edit automata was a first enforcement technique that proposed not only to monitor and halt the system executions but also to transform them: edit automata are able to insert new actions to the execution of suppress them (with a possibility to memorize them for later use). Having this power of modifying program actions at run time, edit automata are provably more powerful than security automata and enforce a class of renewal properties that is shown to be strictly bigger than the class of safety properties.

It was proved in [51] that every decidable renewal property can be enforced by a kind of edit automaton that outputs the longest legal prefix of the input. The renewal property, similar to the liveness property, implicitly assumes that if an infinite sequence is legal then for every prefix of this sequence the liveness holds, or "nothing irremediably bad happens in any finite prefix". It is obviously implied by the fact that an infinite-length legal execution must have an infinite number of legal prefixes. Therefore if an infinite-length execution has something irremediably bad happened in a finite prefix, then the number of valid prefixes is finite, and hence this execution is invalid.

2.3.2 Bounded History Automata

The next attempt to characterize the security policies enforceable by runtime monitors depending on the information stored in the memory of the monitor was done by Talhi et al. [61]. They proposed *Bounded history automata (BHA)* that bounds the automata by the limited history. As such, they distinguish two subclasses of BHA: bounded security automata (BSA) and bounded edit automata (BEA). We will cover the latter case here, since it is a sequence transformer.

The states of BEA represent a bounded history of valid execution sequences: more concretely, every state contains a prefix of the execution sequence already accepted by the automaton and a suffix that is currently suppressed. When a maximum size of history is k, respected BEA is called k-BEA, and the enforcement power of Bounded Edit Automata is bounded by this number. The authors have also proved that for $k, k' \in \mathbb{N}$, when $k < k'$: k'-BEA are more powerful than k-BEA.

Chabot et al. [23] proposed to synthesize security automata from the security properties expressed as Rabin automata. They provide a construction from safety properties in general case and more than safety in case when additional information about the program is obtained by a static analysis. However, the comparison of enforceable properties with other (renewal, liveness) properties is left aside.

An interesting direction was also taken by Khoury and Tawbi [47, 46]. The authors proposed to define an equivalence relation between the original execution of the target and the result of the transformation of that execution by the enforcement mechanism. Their definition of equivalence relation is very generic and cannot guide the user to synthesize better enforcement mechanism.

2.3.3 Synthesis of Enforcement Monitors

Another line of work is concerned the synthesis of runtime enforcement monitors. A great part of the work in this direction is done by Martinelli and Matteucci [54]. They have shown how to synthesize such execution monitors. 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. In a later work [55] the authors generalize the approach in the context of real-time systems.

2.3.4 Generic Enforcement Monitors

Falcone et al. proposed a technique to produce an enforcement monitor given a security property specified as Streett automaton [34, 35]. Parameterizing this automaton, the authors specify four different types of security properties according to safety-progress (SP) classification of properties [25, 24]: safety, guarantee, response and persistence. They give informal definition of these properties[35]:

- *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.

As we can see, safety properties are exactly those that we described earlier in Section 2.1. Since response property guarantees that whenever a sequence is valid, it should have an infinite number of valid prefixes, we can conclude that response property coincides with the renewal property. This is our observation, however the authors of the paper did not explicitly compare their properties with renewal and liveness properties. We will not try to make this comparison but rather pay our attention to the construction of the enforcement mechanism and its way to enforce given properties.

The safety-, guarantee-, and response-transformations describe the semantics of the enforcement mechanisms. All these mechanisms are sequence transformers that always output the *longest valid prefix* of the original execution sequence. In the Chapter 4 we will discuss the relation between these kinds of enforcement mechanisms and other sequence transformers.

In their next work [36] the authors propose an upper-bound of the set of enforceable properties. They get this bound by characterizing properties independently from the enforcement mechanisms that only should comply with soundness and transparency. However, the authors define a property to be enforceable only if each incorrect infinite sequence has a finite number of correct prefixes, and this is a definition of renewal property.

2.3.5 Enforcement Mechanisms for Usage Control

Pretschner et. al. [57] have provided a model of consumer-side enforcement mechanisms for distributed usage control. This model can be used to formally check if a set of mechanisms is able to enforce a given obligation and to check interference of mechanisms. The four classes of mechanisms were presented: inhibition, delay, modification and execution. *Inhibition* prevents specific events from happening, which is similar to the suppression operation of edit automata. *Delay* mechanism postpones the decision about the sequence of events for a fixed number of steps. *Modification* mechanism is able to make a replacement of events, which is similar to two steps of an edit automaton: suppression and insertion. The difference is that the modification mechanism is claimed to be able to cater for concurrent events by replacing one event by the set of events. *Execution* mechanisms can add some events to the execution so that the resulting execution will satisfy the given policy. The ability of adding events is similar to the insertion operation of edit automaton. Hence, they preserve the notion of soundness. However it is not clear how the property of transparency can be checked. The execution mechanism can add arbitrary events, as long as respective properties are not violated. The authors require the effect of applying their mechanisms to be minimal.

2.3.6 A (hidden) assumption

Runtime enforcement mechanisms as sequence transformers (including those that will be presented in this thesis) use one important assumption that *all the actions can be suppressed or modified at runtime.* In practice this is not always possible.

Let us consider a system that executes actions that depend on the results of previous actions. For example, we can consider a Service-oriented architecture (SOA) where services are exchanging messages, for example one service can invoke another service. All the intercommunications between the services are happening through the Enterprise Service Bus (ESB) [26].

In order to enforce some security policies in SOA, an enforcement mechanism is needed to intercept the messages and decide whether the behavior of the system complies with the security policies. One way to implement such an enforcement mechanism is xESB [40], an enhanced version of an ESB. xESB intercepts the messages and makes a decision, whether to accept, delay or modify the messages depending on the security policy. Let us show one motivating example of security policy from the original paper [40], where a hypothetical company "Foo.uk", providing VoIP-based services using a communication platform implemented as a SOA is using an ESB.

Example 2.3.1 Process collect calls only after destination has agreed to pay. *The VoIP destination must have accepted to pay for the call before replying to collect call request.*

The relevant messages on ESB for this policy are:

- i_p for invocation of payment
- r_p for positive response on payment
- i_c for invocation of the collect call
- r_c for positive response of the collect call

The policy specifies that these messages are allowed to be sent in a particular order, that can be described as a regular expression $(i_p, r_p, i_c, r_c)*$ (only after the destination gave a positive response on payment, the collect call can be invoked). This particular scenario requires that all invocation messages are accepted by the enforcement mechanism because if the enforcement mechanism blocks them, the response to those invokations will never arrive. Generally speaking, whenever an acceptance of some messages depends on the acceptance of previous messages, the policy has the form (in the example above n=2):

$$(i_1, r_1), (i_2, r_2), \ldots, (i_n, r_n)$$

Then the enforcement mechanism should accept all sequences of messages the form

$$(i_1, r_1), (i_2, r_2) \ldots, i_j \text{ for all } 1 \leq j \leq n$$

However, this output does not satisfy the security policy being enforced, which contradicts the meaning of enforcement.

In this thesis we make an assumption that holds for all the other state-of-the-art runtime enforcement mechanisms mentioned above:

Assumption 1 *All the actions intercepted by the enforcement mechanism can be suppressed or modified, and this will not influence the other actions emitted by the system*

2.4 Runtime Enforcement Mechanisms for Information Flow Security

There is a large body of related work on information flow security enforcement mechanisms. There are two major approaches to information flow security enforcement: static techniques and dynamic techniques. In particular, we will describe the main approaches to enforcement of a particular information flow security policy, called *noninterference*. Noninterference for programs means that a variation of confidential (high) input does not cause a variation of public (low) outputs.

2.4.1 Static Information Flow Analysis

Sabelfeld and Myers [58] survey static techniques for information flow enforcement. One of the main static techniques is to build a security type system for a particular programming language. A security type system is a set of typing rules that describe what security type is assigned to a program. Whenever a given program is typable with respect to this security type system, the program is proven to be noninterferent. Another static technique is based on programming language semantics. It defines relations on the states of the program execution with respect to the security level: at high security level all the variables are visible to an observer of program execution, and at low security level only low (public) variables are visible to an observer. Noninterference defines that whenever two program states agree on low level variables (are not distinguished by a low level observer), then the outputs of this program are also not distinguishable by a low level observer.

2.4.2 Dynamic Information Flow Analysis

Static analysis techniques have one major drawback: they accept the program only if all its executions ensure noninterference. In this thesis we are more interesred in dynamic techniques, since runtime enforcement mechanisms are dynamic and that should obey similar formal properties.

Automata-based Condentiality Monitoring Work by Le Guernic et al. [44, 43] proposes a monitor that analyses only actual executions of the program. The model of a monitor is an automaton that produces outputs on an input sequence formed by program execution. This automaton is a kind of edit automaton which is able to enforce non-interference of a program with high and low program variables. The states of this automaton contain information about variables that may have been influenced by the initial values of high variables and tracks variety in the context of the execution.

Secure multi-execution Another novel dynamic technique is *secure multi-execution* proposed by Devriese and Piessens [30]. This enforcement method is able to enforce non-interference for a security lattice containing any amount of security levels. The idea of this technique is to execute the original program one time per each security level, while filtering the inputs and outputs of the program in a particular way such that no higher inputs can actually flow into lower outputs.

Formal Properties Similar to the fact that an edit automaton can potentially enforce the same security property in different ways, different dynamic techniques can enforce non-interference in different ways as well. To make a brief discussion, we compare the techniques by Le Guernic et al. and Devriese and Piessens. Formally, we can compare the properties of these mechanisms that were proven in the corresponding papers.

Both of the approaches are proven to be sound, meaning that the output of the two mechanisms form only non-interferent executions. Transparency is a more interesting notion: it means that if the original program execution is non-interferent, then after applying enforcement techniques, the resulting execution is preserved. The problem of noninterference is neither dynamically nor statically decidable. Hence, there is no enforcement mechanism that is transparent, a weaker notion of precision can be proven for mechanisms enforcing noninterference.

Le Guernic et al. [44, 43] proposed a pseudo-trasparency, using a security type system similar to the one of Volpano et al. [63]. The executions of type-safe programs are not modified by the monitoring mechanism, but moreover, other non-interferent executions are not modified by the monitoring mechanism, even though it was not proven formally. Here we quote Le Guernic [43]:

> To show that the inclusion is strict, consider the following program: `x := h; x :=0; output x`. `h` is the only secret input. Every execution is noninterfering. But as the type system is flow insensitive, this program is ill-typed. However, the monitoring mechanism does not interfere with the outputs of this program while still guaranteeing that any monitored execution is noninterfering.

Devriese and Piessens [30] have proven a more interesting result: their enforcement technique is precise in the sense that terminating runs of every termination-sensitive noninterferent program is not modified by an enforcement mechanism. The authors claim that the set of termination-sensitive noninterferent programs strictly includes all programs which are type-safe according to the type system by Volpano et al. [63].

Practically speaking, the monitoring mechanism by Le Guernic et al. seem to be more efficient in practice since it evaluates only one execution at time, while the technique by Devriese and Piessens uses several executions of the program together. On the other hand, the monitoring mechanism operates only with two security levels, while secure multi-execution can handle an arbitrary number of security levels and hence, it is possible that two-execution of the original program can have comparable performance with respect to the monitor of Le Guernic et al.

2.5 Summary

In this chapter we have described several types of runtime enforcement mechanisms: execution sequence recognizers are able only to accept of reject the execution sequence based on the security policy they are enforcing, hence they are only able to halt the execution; execution sequence transformers are more powerful enforcement mechanisms because they are able not only halt, but also modify the tentative execution at runtime. However, these mechanisms make one important and hidden assumption about the original system (see Assumption 1) that all the actions of the system can be suppressed or modified and this does not influence the rest of the system execution.

Runtime enforcement mechanisms as execution transformers are also capable of enforcing more complex security policies, like information flow security policy. In this chapter we also discuss the state of the art literature on the dynamic techniques for information flow analysis.

Chapter 3

Concrete Problems of Runtime Enforcement

In this chapter we discuss the gap that we found in the theory of security policy enforcement mechanisms. We will present several examples of security policies and describe how existing mechanisms can enforce them at runtime.

Different security properties can and should be enforced in different ways. As we know from the state-of-the-art (see Chapter 2), safety property ("nothing bad ever happens") can be enforced by a security automaton, which is a sequence recognizer. This property cannot be enforced in any other better way, since whenever the execution sequence is invalid, it means that all its continuations are invalid as well. So halting the execution as soon as it violates the property is the best way to enforce safety properties.

As we have also discussed in the state-of-the-art, sequence transformers are able to enforce more than safety properties. In particular, edit automata was proven to be able to enforce renewal properties by modification of the execution sequences. In this case it is much more interesting to investigate how exactly the property can be enforced, in other words, what kind of modification should be made to the original execution sequence.

3.1 A Simple Example from the Literature

We use an example from [7] to show the existence of the problem that has not been addressed by other authors.

3.1.1 Market Policy

Example 3.1.1 (Verbatim from [7]) *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).*

3.1.2 Possible Executions

Looking at the example in English we propose several execution sequences in Table 3.1: some satisfy the policy and some do not. When the sequence is not allowed by the policy, the enforcement mechanism should change the sequence in such a way that it becomes legal. In this table we partition the sequences in several groups. In the group "Temporarily illegal sequences that can become good", sequences are not yet finished so that we cannot define whether they are legal (which means satisfy the policy from Example 3.1.1) or not, because they could become good later on. For example, in sequence 1 the `take(1)` action can be followed by a `pay(1)` action that can make the whole sequence legal; on the other hand it can be followed by `browse` that will make the sequence illegal.

The group "Legal sequences" contains sequences that satisfy the policy, for example `take(1)`; `pay(1)`.

The initially illegal sequences are divided into two groups. The first group contains the sequences with an initially bad prefix but the suffix can become legal later. For example, sequence 6 has an illegal prefix `take(1) browse`, however the suffix `pay(2)` can be extended to legal sequence since it can be a beginning of sequence 5: `pay(2)`; `take(2)`. The second group contains sequences that have an illegal prefix followed by a

Table 3.1: Sequences of actions for market policy

Temporarily illegal sequences that can become good

No	Sequence of actions	Expected output
1	take(1)	·
2	pay(2)	·
3	pay(2); browse	browse

Legal sequences

No	Sequence of actions	Expected output
4	take(1); pay(1)	take(1); pay(1)
5	pay(2); take(2)	pay(2); take(2)

Initially illegal sequences, but a later suffix can become good

No	Sequence of actions	Expected output
6	take(1); browse; pay(2)	warning
7	take(1); pay(2)	warning

Initially illegal sequences with legal continuation

No	Sequence of actions	Expected output
8	take(1); browse; pay(2); take(2)	warning; browse; pay(2); take(2)
9	take(1); pay(2); take(2)	pay(2); take(2)
10	pay(1); browse; pay(2); take(2)	browse; pay(2); take(2)

legal continuation, such as sequence 8: it has an illegal prefix `take(1)`; `browse`, but its suffix is a legal (sequence 5).

There are some other sequences like `pay(1)`; `browse`; `pay(2)`; `take(2)`; `take(1)` that we could not add to neither of the groups. This happens because the text leaves open 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.

3.1.3 Problems in Enforcement

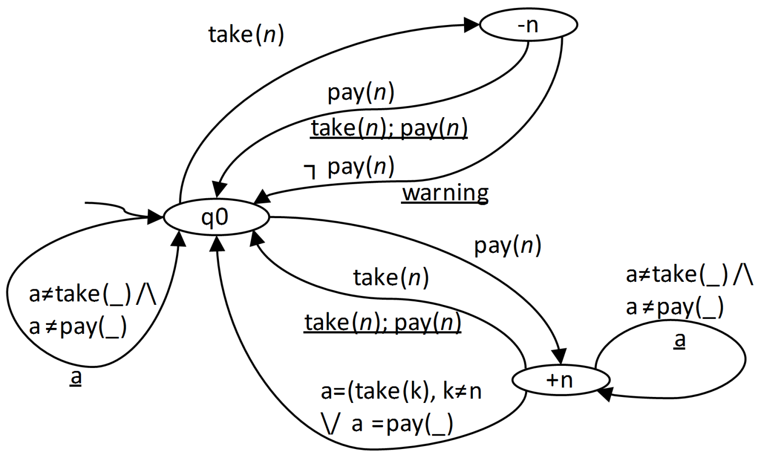
The original edit automaton proposed in Figure 2 of [7] is shown in this thesis in Figure 3.1. The graphical notation used in this figure is the same as in [7] with only one addition: for readability we underline the output at the transition. The nodes in the picture represent the automaton states and the arcs represent the transitions. The action that is above the arc defines an input action. Output actions are underlined and placed below the arcs. Arcs with no underlined sequence represent transitions where there is no output. If there is no transition for the given state and action, then the automaton halts.

Let us look at sequence 4 from the Table 3.1: `take(1)`; `pay(1)`. The edit automaton starts in the initial state $q0$. When it reads the first action `take(1)`, it moves to the corresponding state $-n$ and waits for the next input action to arrive. When the action `pay(1)` arrives, the automaton moves back to the state $q0$ while outputting the `take(1)`; `pay(1)` sequence (as indicated at the output of the arc from $-n$ to $q0$).

The policy given in Example 3.1.1 can be enforced by edit automata in different ways. One of the notions that guarantees enforcement of the policy is called *effective$_=$enforcement*. This notion ensures that all legal sequences are not changed by the edit automaton, while all illegal sequences are changed to *some* legal sequences or to empty sequences. The automaton in Figure 3.1 provides effective$_=$enforcement of the given policy since it does not change the legal sequences and always changes illegal sequences to some legal ones. In the following, we will use the predicate called *property* that decides whether the sequence is legal or not.

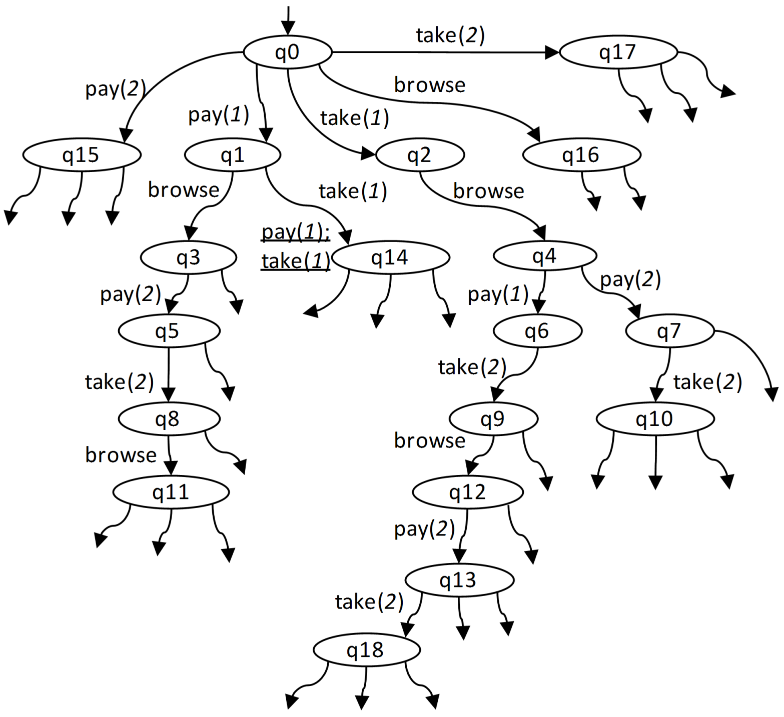
Theorem 8 [7] says that any renewal property can be effectively$_=$enforced by an edit automaton. The proof of this theorem presents a construction of an edit automaton from a given renewal property. We briefly sketch the main idea of the construction from the proof of Theorem 8 [7] in the following steps:

- States of the edit automaton contain two finite sequences: sequence of actions seen so far and the sequence of actions that were suppressed
- Consider processing a new input action:
  - in case the sequence seen so far becomes valid, the automaton inserts the suppressed sequence followed by a new input.
  - in case the sequence seen so far does not become valid, the automaton suppresses the current input action and adds it to the suppressed sequence in the next state.



This edit automaton was originally presented in Figure 2 of [7] and it is effectively$_=$enforcing the market policy.

Figure 3.1: Edit automaton verbatim from [7].



This edit automaton is constructed following the proof of Theorem 8 [7]. It is easy to see that this automaton effectively$_=$enforces the market policy, but has an infinite number of states and provides different kind of enforcement for the same policy.

Figure 3.2: Edit automaton reconstructed by the proof of Theorem 8 from [7].

Table 3.2: Difference in output for edit automata

No	Input	Output	
		Edit automaton from Figure 2 of [7]	Constructed edit automaton by Theorem 8 [7]
4	take(1); pay(1)	take(1); pay(1)	take(1); pay(1)
6	take(1); browse; pay(2)	warning	·
8	take(1); browse; pay(2); take(2)	warning; pay(2); take(2)	·
9	take(1); pay(2); take(2)	warning	·
10	pay(1); browse; pay(2); take(2)	browse	·

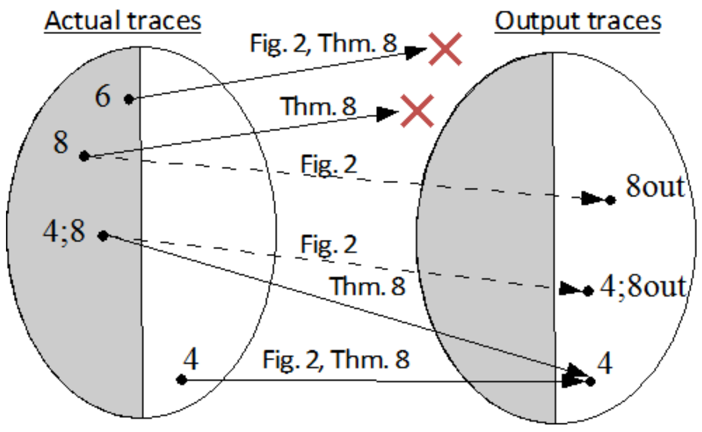
We will give a more detailed construction verbatim from the proof of Theorem 8 [7] in Section 4.3.

For sake of simplicity, we show a constructed edit automaton only partially, for the set of actions {`take(1)`, `take(2)`, `pay(1)`, `pay(2)`, `browse`}. Here we use a `browse` action just to present some other actions that the user can do after paying before taking the apples. According to the text, an action `warning` is considered to be an output action in the edit automaton in Figure 2 of [7] (see Figure 3.1). 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. In Figure 3.2 we follow the construction from the proof of Theorem 8 in order to build the edit automaton.

The edit automaton from Figure 2 of the original paper (Figure 3.1) and the one constructed by the proof of Theorem 8 of the same paper (Figure 3.2) actually produce different output for the same input. In Table 3.2 we show some cases of input and output of both automata.

Sequence 4 is legal, hence it is not changed by both automata. Sequence 6 has illegal prefix and not yet legal suffix, hence both automata halt (let us ignore the `warning` action since it can be easily added to the construction from Theorem 8). However, for other bad sequences that have different nature, the automata behave differently. The output set of the automaton from Figure 2 of [7] is larger than the output of the automaton from Theorem 8 [7].

The "strange" behavior begins when we take the sequences from the group "Initially illegal sequences with legal continuation", like sequence 8. The edit automaton from Figure 2 of [7] can skip the illegal prefix `take(1); browse` and output the legal suffix



Grey area is the area of invalid traces, white area shows valid traces. Edit automaton originally from Figure 2 [7] (that we show in Figure 3.1) and edit automaton constructed following the proof of Theorem 8 from the same paper produce different output for the same invalid execution sequences.

Figure 3.3: Relation between input and output for edit automaton originally from Figure 2 [7] and Theorem 8 [7].

`pay(2); take(2)`. However, the automaton from Theorem 8 halts as soon as an illegal prefix does not have a legal continuation. In case of sequences 9 and 10 both automata produce a strange output in a sense that the agent has paid but never got any apple.

Analyzing Table 3.2, 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. When the sequence is illegal the output of both edit automata is unexpected and potentially different for each automaton.

3. The edit automaton constructed following the proof of Theorem 8 [7] is a very particular kind of the edit automaton.

In Figure 3.3, we show the relation between input and output for edit automaton from Figure 2 of [7] and edit automata from Theorem 8 [7] with respect to the "good" and "bad" traces. By 4;8 we mean the concatenation of the sequence 4 with the sequence 8.

By 8out we mean an output sequence of Figure 2 automaton when processing sequence 8 as input. It is shown in Table 3.2.

As we can see, both automata from Figure 2 of [7] and Theorem 8 [7] are edit automata, and they both effectively$_=$enforce the market policy. *Why these two edit automata are producing different outputs?*

We shall investigate this question in this thesis and by doing so we will solve Issue 1. We will analyze different classes of edit automata that explain the behavior of the edit automata from Figures 3.1 and 3.2 and formally define relation between them in Section 4.3.

Another question is *How to construct a runtime enforcement mechanism that enforces given security policy in a desirable way?*

We raise this question because as we have seen on the example of market policy, outputting the longest valid prefix is not always the best way of enforcing the security property. We shall address this question to solve Issue 2.

Before proceeding to the classification and rigorous formal analysis of security properties and their enforcement in later chapter, let us present the main example from industry that we will use in this thesis.

3.2 A Case Study from Industry

This example also represents a property that is neither pure safety, nor liveness and allows sequences to alternate between being valid and invalid.

3.2.1 Drug Dispensation Process

We propose an example based on a healthcare process of drug dispensation. In Italy hospitals accredited with the Public National Health Service are in charge of administering drugs and providing diagnostic services to patients. Usually in these (public or private) hospitals there is a generic *dispensation process description* that allows hospitals to refund the drugs administered and/or supplied in the hospitals' outpatient departments to the patients that are not hospitalized. In particular, there is a process called *File F* that allows refunding of the drugs for specific critical and chronic diseases. This refunding is usually done by the public authority. As another example, if the patient is using a specific drug for the research program purposes (i.e. the patient has been enrolled for the clinical trial for the testing of that drug) then the reimbursement should be done by the clinical trial funds.

Drug dispensation process is a high level business process. It involves human participants as well as IT technologies. Some of contained tasks are completely human activities without any interaction with IT system (e.g. all patients tasks, or delivering drugs from stock to patient (physically) by doctor or nurse). Hence, some of the activities may be done in a different way than described in the default process, and we will make a precise distinction between the actual execution of the process and the official default process. The described model of drug dispensation assures that if all the activities adhere to it, they comply with the encoded rules (e.g. File F process) and the drugs will be refunded.

In Figure 3.4 we present a simplified version of the BPMN (Business Process Model Notation) diagram of drug dispensation process.[1]. We show the whole drug selection process in the upper part of the figure and emphasize its drug selection subprocess in the lower part of the figure. We will discuss this subprocess in more details in the sequel.

The drug dispensation process starts when the Patient brings his prescription sheet to Doctor or Nurse (we will say Doctor from here on). We will describe this process step-by-step:

1. The first step of the process is "Retrieve Doctor's Data". The Doctor authenticates himself by entering his ID. The system identifies Doctor's operational unit and enables Patient identification.

2. Next the Doctor runs the subprocess "Identify Patient", we will not describe the details of this subprocess since they are not important for this running example.

3. When Doctor made an identification, he asks the Patient whether Patient requires anonymization of his personal health records ("Check Anonymization").

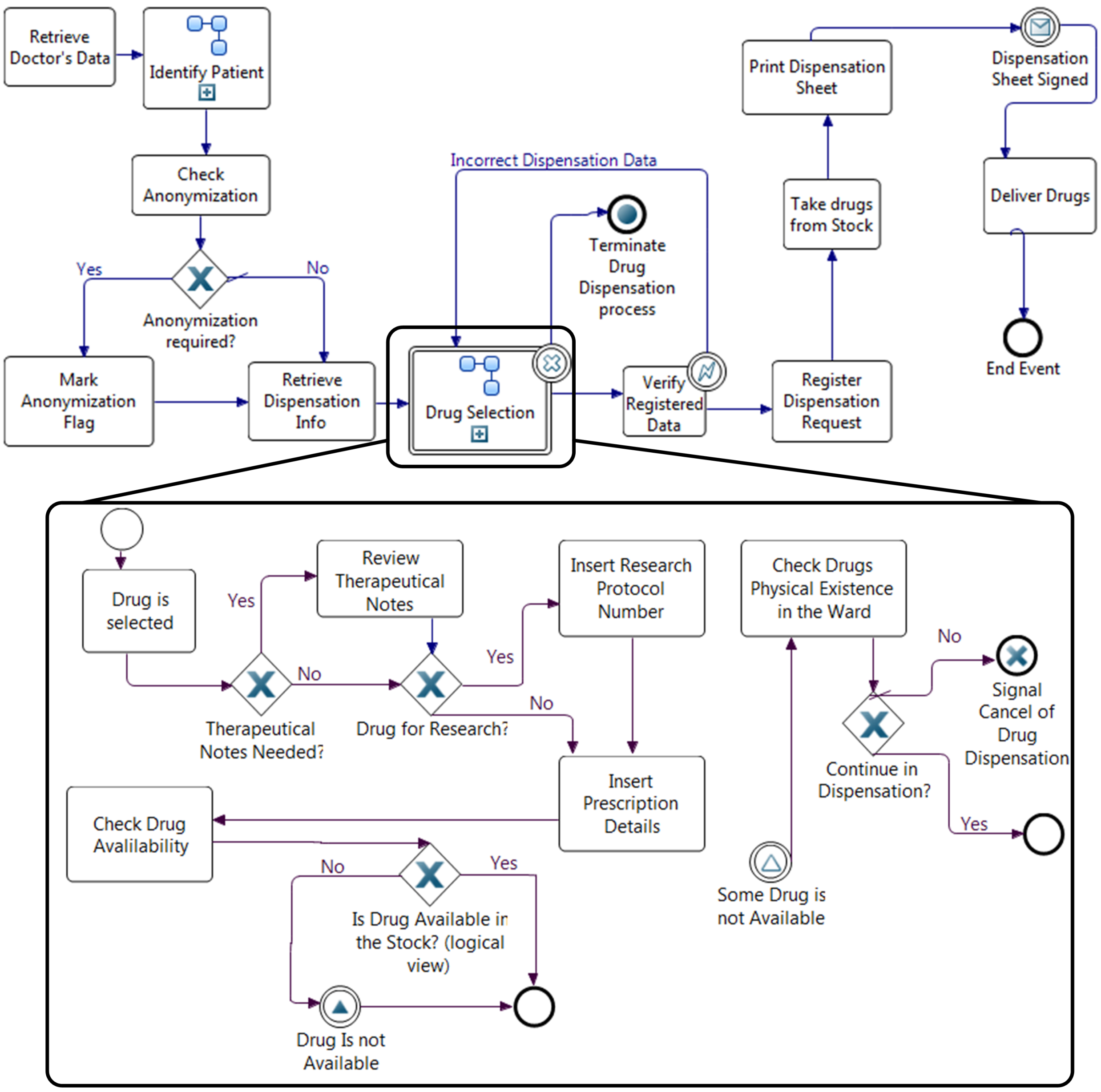
4. The Doctor marks that it should be anonymized it if required ("Mark Anonymization Flag").

5. The system retrieves all necessary data for selecting the drugs for dispensation and offers the option to select drugs to the Doctor ("Retreive Dispensation Info").

6. "Drug Selection" is a special subprocess that we will describe in details later.

7. After drug selection the Doctor performs a number of steps. He verifies candidate drug list (for dispensation) and continues or restarts the process of drug selection. Then Doctor registers the dispensation request, takes the drugs physically from the

[1]We would like to thank ANECT (`http://www.anect.com/en/`) for developing original BPMN diagrams of the full drug dispensation process.


Retrieve Doctor's Data
Identify Patient
Check Anonymization
Yes
No
Anonymization required?
Mark Anonymization Flag
Retrieve Dispensation Info
Incorrect Dispensation Data
Drug Selection
Terminate Drug Dispensation process
Verify Registered Data
Register Dispensation Request
Take drugs from Stock
Print Dispensation Sheet
Dispensation Sheet Signed
Deliver Drugs
End Event
Drug is selected
Yes
Review Therapeutical Notes
No
Therapeutical Notes Needed?
Yes
Drug for Research?
No
Insert Research Protocol Number
Insert Prescription Details
Check Drugs Physical Existence in the Ward
No
Signal Cancel of Drug Dispensation
Continue in Dispensation?
Yes
Some Drug is not Available
Check Drug Avalilability
No
Yes
Is Drug Available in the Stock? (logical view)
Drug Is not Available

The upper part of the figure shows the BPMN diagram for drug dispensation process. Subprocess "Drug selection" represents the sequence of actions when doctor selects a drug for a patient.

Figure 3.4: BPMN diagram drug dispensation process

stock, prints dispensation sheet, brings drugs to the Patient and archives copy of dispensation sheet, signed by Patient.

This process was implemented in the MASTER project and more details of the process can be found at [53].

As a running example for this thesis, we will use the drug selection subprocess. Execution of this subprocess is repeated by the Doctor for every drug in the prescription. We describe it step-by-step:

1. The Doctor selects one drug from the candidate drug list for his Patient ("Drug is selected").

2. If the drug is highly sensitive, reviewing therapeutical notes is needed ("Review Therapeutical Notes"). In this case they will be shown to the Doctor.

3. The system checks drug's submission to Research program and in case the drug is registered shows the notification to the Doctor. In case Doctor receives such notification, he should insert the research protocol number, a number of the protocol according to which the drug can be given to the Patient ("Insert Research Protocol Number").

4. The Doctor performs "Insert prescription details" for the drug he selected for his Patient.

5. The system checks drug availability in stock and eventually notifies the Doctor.

6. If the drug is not available in stock Doctor checks the physical existence in the ward and then decides to continue or cancel the dispensation process ("Check Drugs Physical Existence in the Ward").

To ease the comparison with other papers on runtime enforcement [7, 61] and to the example we have described in Section 3.1, we present the BPMN process using finite state automaton. This representation is natural, since from a point of view of the Doctor a workflow described above is a sequence of actions that should be performed. To simplify the translation we assume that each choice in BPMN diagram corresponds to the action in resulting process execution that communicates the choice, e.g., if the drug is for research then the corresponding action is "Drug is for research", if the drug is highly sensitive, then an action "Therapeutical notes needed" will be shown. Not only in our example but in many practical cases the policy is given implicitly by describing the workflow corresponding to the legal executions, that can be repeated several times. This is precisely the case when we represent a process description using automaton. Formal definition and a corresponding automaton will be given later in Chapter 5.

The security policy P described desired sequences of actions that should be done for drug dispensation. We will use the following notations for the actions in the process:

Dis	Drug is selected
Tnn	Therapeutical notes needed
Rtn	Review therapeutical notes
TnNn	Therapeutical notes Not needed
Dr	Drug is for research
Irpn	Insert research protocol number
DNr	Drug is Not for research
Ipd	Insert prescription details
DNas	Drug is Not available in the stock
Dpew	Drug physically exists in the ward
Das	Drug is available in the stock

The execution sequences consist of combinations of these actions, and the security policy specifies only those sequences that do not violate the process description. To simplify the description, Dpres is either equal to Ipd; Das or to Ipd; DNas; Dpew.

Example 3.2.1 *The security policy P consists of the following traces:*

SimpleRun Dis*;* TnNn*;* DNr*;* Dpres*,*

NoteRun Dis*;* Tnn*;* Rtn*;* DNr*;* Dpres*,*

ResearchRun Dis*;* TnNn*;* Dr*;* Irpn*;* Dpres*,*

NoteResearchRun Dis*;* Tnn*;* Rtn*;* Dr*;* Irpn*;* Dpres*,*

and their closure under concatenation: for every $\sigma, \sigma' \in P$: $\sigma; \sigma' \in P$. Notice that an empty trace **NoRun** *also satisfies the policy.*

3.2.2 Possible Executions

Here we present several executions of the drug selection subprocess and analyze the kind of enforcement that the up to date techniques can propose.

Let us assume the following execution of the process, where 3 different drugs are in the prescription list, hence the execution will consist of 3 parts, that we call an iteration, for each drug:

1. The first drug is selected and it is not highly sensitive, so therapeutical notes are not needed; the drug is for research, so the Doctor inserts research protocol number; then he inserts prescription details; and the drug is available in stock. The sequence of actions for this iteration is: Dis; TnNn; Dr; Irpn; Ipd; Das, which is a sequence denoted by ResearchRun, and it is compliant with to the policy P.

2. Then for the second drug: the Doctor selects it; the drug is not highly sensitive so therapeutical notes not needed; the drug is for research but the Doctor inserts prescription details only; the drug is available in stock. The sequence of actions for this iteration is: Dis; TnNn; Dr; Ipd; Das, which is a sequence not accepted by the policy P. Indeed, in this part of the process execution the drug is for research but the research protocol number is not inserted.

3. The third part of the execution consists of actions: the Doctor selects a highly sensitive drug, so therapeutical notes needed; the Doctor reviews therapeutical notes; the drug is not for research; Doctor inserts prescription details, and the drug is available in stock. The sequence of actions for this iteration is: Dis; Tnn; Rtn; DNr; Ipd; Das. This iteration is denoted by NoteRun and is compliant with the policy. Hence, this part is correct.

3.2.3 Problems in Enforcement

Let us discuss how the drug dispensation process can be enforced by existing runtime enforcement techniques. We had a long discussion about the problems specified in this section with the colleagues from the Hospital San Raffaele (HSR), Milan, Italy. Some issues are discussed in a joint publication [17][2].

More than the longest valid prefix

The execution consisting of three iterations has to be changed in order to become compliant. The only construction we have seen in the state-of-the-art papers is in the proof of Theorem 8 in [7]. The idea behind this construction is to monitor the actions of the target one by one and suppress them in case the sequence seen so far is invalid. As soon as the target is going to execute a new action that validates the sequence seen so far, the constructed edit automaton will insert all the suppressed actions and this new action.

[2]Disclaimer: The views expressed in this thesis are those of the author and do not reflect the official policy of the Hospital San Raffaele

Hence, the result of this construction is an edit automaton that always outputs the longest valid prefix of the tentative target execution.

Such edit automaton for our drug dispensation process example will suppress the actions of the first iteration until the last action arrives, i.e. actions Dis, TnNn, Dr, Irpn and Ipd will be suppressed. As soon as the action Das arrives, the sequence of actions seen so far will comply with the policy P and so the edit automaton will output this suppressed sequence that we denoted by ResearchRun. After that the automaton will suppress the actions of the second iteration: Dis, TnNn, Dr, Ipd, Das. The automaton is constructed in such a way that it will wait for arrival of a new action that validates this iteration. However, such action does not exist because the doctor already made an irremediable error: he did not insert the research protocol number (corresponding action Irpn) when the drug is for research (action Dr happened). The third iteration, that is valid by itself, will never be output by this automaton because it will suppress all the following actions after the second iteration waiting for this iteration to become valid, which will never happen.

Even if we accept the idea that an incorrect execution should be dropped, the acceptable behavior for the administrators of the e-health system is just to drop the second part of the execution. We are going to propose the solution to this problem in Chapter 5 and will address in this way Issue 2.

More than effective$_=$enforcement

Similar to drug dispensation process, italian hospitals must guarantee the compliance of many business processes involving large amounts of money (reimbursements from public health authorities for drugs dispensation), significant privacy concerns (drugs can be related to HIV or other serious illnesses), major safety considerations (drugs might have serious side-effects), and compliance with many regulations. These regulations can change frequently and a runtime enforcement mechanism could guarantee the compliance of each process with minor efforts.

Unfortunately, an enforcement mechanism must offer some guarantees to the hospital on what happens when things are not according to the policy. At present the only offered formal guarantees is called effective$_=$enforcement that comprises the notions of soundness and transparency. The latter two notions are currently used in the state of the art literature on runtime enforcement [22, 35, 52, 61, 64]. As we have seen in the example of market policy, this notion is necessary but not sufficient. It is also not sufficient in case of drug dispensation process since it allows outputting the longest valid prefix of the execution when another kind of enforcement is preferred.

If we think about practical aspect of enforcement, what the risk manager of the hospital wants to know is "What the enforcement mechanism *normally* does when a doctor's action does not respect the policy?" Does it abort the whole transactions if a research protocol number is not entered (like Security Automata [59])? Does it alert the head of the department if we are prescribing a drug out of stock (Usage control enforcement mechanism [57])? Does it wait till the doctor opens the therapeutical notes before committing the transaction to the audit logs (Longest-valid-prefix automaton [51])?

These questions show again that Issue 3 is an important problem. There is no general, principled guarantee that the illegal executions are transformed in a way that the end users expect. The notion of effective$_=$enforcement does not provide such guarantee. We will address this problem in Section 6.1.

Error toleration

Another challenge in policy enforcement is the trade-off between writing simple policies and enforcing complex run-time behaviors. This is particularly important for workflows like the drug dispensation process, in which human actors must interact with the policy enforcement mechanism.

The actual execution of the process must comply with File F process (so that drugs will be reimbursed by correct parties), as well as with other health and privacy regulations. Hence, from the perspective of the runtime enforcer, it is a better case if the policy describes all the rules and regulations that the process executions should be compliant with.

Unfortunately, from the perspective of policy management, detailed policies are difficult to write, are difficult to check for consistencies, and de facto impossible to communicate to the end users. So, the simpler the policy the better. In our case a policy is a simple drug dispensation process that is, in its essence, a linear sequence of steps (with few loops for stock replenishing). This is easily understood by doctors and nurses. However each and every step might be subject to many exceptions or common errors (for example closing a window instead of pressing the button done). Detailing and representing all these exceptional steps in a graphical form would make the protocol unreadable.

In our domain there is a further difficulty: our users would definitely insist that there is only *one policy*, i.e. the "official" protocol workflow. There is no such a thing as a policy including all exceptions (this would require validation by the risk manager and the responsible person for the pharmacy and dispensation process).

There is another problem in the enforcement of processes. During the executions doctors, nurses, and pharmacists would not like to be disturbed in their primary mission (delivering the right drug to the right patient) because an insignificant deviation from the default workflow has taken place. In the current mechanisms, runtime exceptions and the consequent additional workload needed to complete the execution distract users and convince them that “The system doesn’t work”. If such disruptions occur too often users will increasingly try to bypass the runtime enforcer.

Given this official protocol workflow, the state of the art literature does not propose a construction of enforcement mechanism that can tolerate some number of exceptions or common errors that the users may make during the process executions. Up to now, the runtime enforcement theory provides only strict mechanisms that would not allow any deviations from the original protocol and rather stop the process execution than tolerate any error.

Hence there should be a generic algorithm for constructing runtime enforcement mechanisms for such systems, where insignificant errors can take place. This problem corresponds to Issue 4 that we will address in Section 6.2.

3.3 Summary

In this chapter we have presented two case studies that will be used in the rest of this thesis to compare different enforcement mechanisms.

A case study from the literature is a market policy, that was originally presented in [7]. This is a simple policy, but even for this policy the problem of enforcement is important.

We have also presented a case study from industry: a drug dispensation process from the hospital describes correct actions that should be done by a doctor in order to comply with the regulations and File F process, such that the drugs in the end are reimbursed by the correct parties. The enforcement of this policy also requires discussion and will be presented later in the next chapters of this thesis.

Chapter 4

Classification of Properties and Mechanisms

This chapter provides two major contributions. First, we present the existing classification of security properties and provide a formal definition for a new kind of security property. This new property is discussed and the relations to the other properties are shown. Second, we address the problem defined in previous chapter and show the relations between enforcement mechanisms that are enforcing the same security property in different ways.

4.1 Notations

In all the formalizations we will use the standard notation in the theory of runtime enforcement mechanisms [7, 38, 59]. The target is specified at a high level of abstraction, where Σ is the set of all possible target actions that are relevant for the security policy (also known as security-relevant actions). An *execution sequence*, or a trace, is a finite or infinite sequence of actions; a finite sequence indicates a terminating target execution, while the infinite one indicates the nonterminating execution of the target. The set of all finite sequences over Σ is denoted by Σ^*, the set of all infinite sequences is Σ^ω, and the set of all (finite and infinite) sequences is Σ^∞. Execution sequences are denoted by σ or τ.

The symbol $\cdot$ denotes an empty sequence. By $\sigma[i]$ we denote the ith action in the sequence; $\sigma[..i]$ denotes the prefix of σ involving the actions $\sigma[1]$ through $\sigma[i]$; and $\sigma[i+1..]$ denotes the suffix of σ involving all other actions beside $\sigma[..i]$. We use the notation $\tau;\sigma$ for concatenation of two sequences, where τ must have a finite length.

The notation $\tau \preceq \sigma$, or $\sigma \succeq \tau$ denotes that τ is a finite prefix of (possibly infinite) sequence σ. When τ is a *strict* prefix of σ (i.e. $\tau \preceq \sigma$ and $\tau \neq \sigma$), we write $\tau \prec \sigma$. We write $\forall \tau \preceq \sigma$ as an abbreviation for $\forall \tau \in \Sigma^* : \tau \preceq \sigma$ and $\exists \tau \preceq \sigma$ for $\exists \tau \in \Sigma^* : \tau \preceq \sigma$. Similarly, we write $\forall \sigma \succeq \tau$ as an abbreviation for $\forall \sigma \in \Sigma^\infty : \sigma \succeq \tau$ and $\exists \sigma \succeq \tau$ for $\exists \sigma \in \Sigma^\infty : \sigma \succeq \tau$.

4.2 Classification of Security Properties

We first give the formal definitions of the security properties in the state of the art that we informally described in the Section 2.1. Then we introduce a new class of security properties and show its relation to the other existed classes.

4.2.1 Formal Definitions of Security Properties

As we have discussed, security property is a predicate $\widehat{P}$ for a security policy P. Formally, for all execution sequences $\Pi \subseteq \Sigma^\infty$, the following is true:

$$P(\Pi) \iff \forall \sigma \in \Pi : \widehat{P}(\sigma) \tag{4.1}$$

When a property $\widehat{P}$ holds for an execution sequence, we will say that the sequence is *legal* or *valid*, and when the property does not hold, we will call the sequence *illegal* or *invalid*. We say that an execution sequence σ is *irremediable* with respect to the property $\widehat{P}$ when the following holds:

$$\neg\widehat{P}(\sigma) \land \forall \tau \succeq \sigma : \neg\widehat{P}(\tau) \tag{4.2}$$

There is a one-to-one correspondence between the security property P and its characteristic predicate $\widehat{P}$, hence we will use notation $\widehat{P}$ to refer to the security property and to its characteristic predicate.

There are three major properties in the literature of runtime enforcement mechanisms: safety, liveness and renewal properties.

The *safety* property defines behavior as "nothing bad ever happens", it means that as soon as something bad happens, this is irremediable: if the execution became illegal, it can never become legal again. More formally:

Definition 4.2.1 (Safety property) *A property $\widehat{P}$ is a* safety property *if and only if*

$$\forall \sigma \in \Sigma^\infty : (\neg\widehat{P}(\sigma) \Rightarrow \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \neg\widehat{P}(\tau)) \tag{4.3}$$

Additional to safety properties, there are *liveness* properties that claim that every illegal trace of finite length is not irremediable. In the other words any finite execution can always be extended to satisfy the property.

Definition 4.2.2 (Liveness property) *A property $\widehat{P}$ is a* liveness property *if and only if*

$$\forall \sigma \in \Sigma^* : \exists \tau \succeq \sigma : \widehat{P}(\tau)) \tag{4.4}$$

For example, consider a property $\widehat{P}$ specifying that an execution is legal if eventually an audit is performed which corresponds to an action a in the trace. $\widehat{P}$ is a liveness property since for every illegal finite-length sequence σ there exists a legal continuation $\tau = \sigma; a$.

Ligatti et al. [51] defined a new class of properties called *infinite renewal properties.* Formally, a property $\widehat{P}$ is an infinite renewal property on a target with an action set Σ if and only if the following is true:

Definition 4.2.3 (Infinite renewal property) *A property $\widehat{P}$ is an* infinite renewal property *if and only if*

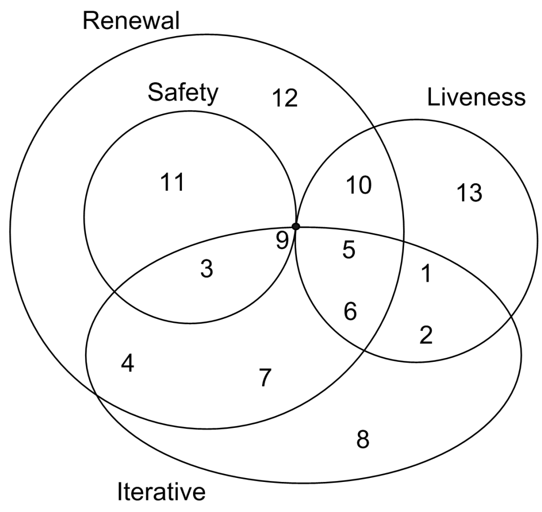
$$\forall \sigma \in \Sigma^\omega : \widehat{P}(\sigma) \Longleftrightarrow (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \widehat{P}(\tau)) \tag{4.5}$$

According to renewal property, every infinite-length sequence is valid if and only if it has infinitely many valid prefixes. This intuition is written above in this way: an infinite sequence σ is valid iff for all its prefixes there exists a valid continuation within σ. Notice that these valid continuations τ are always finite prefixes of σ. For brevity we will call them "renewal properties" in the rest of this thesis.

Having found out that the same security property can be enforced by different edit automata and this enforcement can be done in different ways, we decided to investigate this problem on the example of the security policy in a healthcare domain. Let us come back to the example of drug selection process and define which kind of security property it corresponds to. Here the description of the process is actually a default protocol that the doctor should follow in order to be compliant with the requirements imposed on the drug selection subprocess. A property $\widehat{P}$ that describes this behavior has a particular structure: the execution is legal if it consists of iterations that are in turn valid according to $\widehat{P}$. We generalize this class as *iterative properties* (assuming that empty trace is always valid), a new property introduced in this thesis.

Definition 4.2.4 (Iterative property) *A property $\widehat{P}$ is an* iterative property *if and only if*

$$\forall \sigma \in \Sigma^* : \forall \sigma' \in \Sigma^\infty : \widehat{P}(\sigma) \wedge \widehat{P}(\sigma') \Longrightarrow \widehat{P}(\sigma; \sigma') \tag{4.6}$$



Legend

1 Nontermination	7 Transaction property
2 Resource availability	8 Termination + file access control
3 Stack inspection	9 Trivial
4 Log out and never open files	10 Increasingly longer sequences
5 Property 4 on system without file-open actions	11 At most 100 SMS messages
6 Eventually audits	12 At most 100 SMS and eventually audit
	13 After k transactions send a report

Figure 4.1: Relationships between security properties

4.2.2 Relations Between Security Properties

Figure 4.1 (extending Figure 1 from [51]) represents the relationship between safety, liveness, renewal properties and iterative properties from the point of view of good executions. The original version of this figure contains 9 examples of security properties, while we extend it by adding a new class of iterative properties and by adding one more example to every new intersection of existing classes of properties.

Property 1 is a property of nontermination. It is a liveness property since it holds for all infinite-length traces and it is not a safety property because all the finite prefixes of the good traces always have good continuations. However, since no finite-length traces are valid, they will always satisfy the definition of iterative property: the condition on the left-hand side of the implication will never hold, and hence (even if it's counterintuitive) nontermination can be considered an iterative property.

Property 2 originally described in [51] is a liveness and iterative property. Assume that the system opens some resource i with action o_i and closes it with action c_i. The property 2 claims that all the opened resources must be eventually closed. This property is a liveness property because any illegal sequence can be made legal by adding all the necessary closing actions. However, it is not a renewal property because a valid infinite sequence like $o_1; o_2; c_1; \ldots o_i; c_{i-1}; \cdots$ (where o_i corresponds to an opening action and c_i corresponds to a closing action) does not have any valid prefixes. This property is however iterative, because for any two sequences (where the first one should be finite) that satisfy this property, their concatenation also satisfies it.

Property 3 is a stack-inspection policy that is a particular kind of an access-control policy: an access is granted or denied based on the current nesting of function calls. We will consider a simplified model of stack inspection [39, 42].

Consider the following example: function g has permission to access resource R, but function f does not have this permission. Function f calls function g and g requests resource R. The access should not be granted because f does not have required permissions. In order to decide whether the access should be granted or not, the whole chain of function calls should be examined.

At every function call a new stack frame is created. Each frame contains the local state of the function and the permissions directly granted to it. One of the simplest way to calculate the effective permissions is to take an intersection of the permissions of all functions on the call stack. In the example above, the effective permissions is equal to $perm(f) \cap perm(g)$.

We will say that an execution of the program satisfies the stack inspection policy, if every function call has effective permissions to access the required resource, and at the end of execution the call stack is empty.

The stack inspection policy is definitely a safety property because once the access is not granted based even on primitive stack inspection, it cannot be granted later on because the function in the call stack does not have required permissions. On the other hand, stack inspection policy is an iterative property because whenever all the functions in two call stacks of two executions have required permissions and the first execution finished with an empty call stack, then the concatenation of these two executions again satisfies the stack inspection policy.

Property 4 "Log out and never open files" is a renewal property. Assume the system has the following actions: a_3 ranges over actions for opening files, a_2 over actions for logging out and a_1 over all other actions. The policy says that the user must eventually log out and never open files. So, this property can be written as $(a_1^*; a_2)^\omega$. It is not a safety property because there exists an illegal sequence of only a_1 actions that can be extended to a legal one by adding a_2. It is not a liveness property, because there is an illegal sequence containing a_3 that can never be extended to a legal one. However, it is a renewal property and also an iterative property because the concatenation of every pair of legal sequences produce a legal sequence.

Property 5 is the same as property 4 but on a system that only performs actions a_1 and a_2. The property claims that only sequences of the form $(a_1^*; a_2)^\omega$ are valid. Hence, this property is a liveness property because every invalid finite-length sequence (that apparently is a sequence of the form a_1^*) can be extended to a valid one by adding a_2. It is also a renewal property because every valid infinite-length sequence has an infinite number of valid prefixes of the form $(a_1^*; a_2)^*$. This property is iterative since the concatenation of two valid sequences is always valid.

Property 6 specifies that an execution is good if eventually an audit is performed which corresponds to an action a in the trace. It is obviously a liveness property since every invalid sequence can be extended to a valid one by adding action a in the end. It is not a safety property because an invalid sequence (not containing one) has a valid continuation (containing a). It is a renewal property because an infinite-length valid execution must have infinitely many prefixes in which a appears, and an innite-length invalid execution has no valid prex because a never appears. This property is also iterative, because by concatenating two sequences in which a eventually appears, we get a valid sequence.

Property 7 is called "transaction property". Let τ range over finite sequences of single, legal transactions. A transaction policy is τ^∞ and a legal execution is the one containing any number of valid transactions. This property is not liveness because an invalid transaction (that is different from τ) can never become valid again. It is a renewal property because every valid infinite-length sequence of the form τ^∞ has infinite number of valid prefixes τ^*. It is an iterative property because every concatenation of two valid sequences of the form τ^* and τ^∞ produces a valid sequence.

Property 8 is a combination of a termination property and a property "never access private files". Termination by itself is a liveness property because all finite sequences are valid. The property "never access private files" is by itself a safety property because once a private file was accessed, the sequence can not become valid again. However, the combination of these two properties is neither safety nor liveness. It is also not a renewal property because invalid infinite sequences (where private files are never accessed) have infinitely many valid prefixes. However, property 8 is an iterative property. If we concatenate two legal sequences that are terminated and never access private files then the resulting sequence is also be legal.

Property 9 is a trivial property that considers all sequences legal. This property is an iterative property as well.

All the properties mentioned so far are iterative properties. We know describe the properties that are not iterative.

Property 10 "Increasingly longer sequences" states that the sequence is legal if and only if it is infinite or its length belongs to the following set of numbers $\{F_i\}$: $F_0 = 1$, $F_{i+1} = 2F_i + 1$. Every illegal finite sequence can be prolonged such that its length will belong to the defined set of numbers, so this is a liveness property. It is also renewal because every legal infinite-length sequence has an infinite number of legal prefixes. However, this property is not iterative: by concatenating two legal sequences a new illegal sequence is always obtained.

Property 11 "at most 100 SMS messages per application run can be sent by a mobile device." This property is useful in practice when the use of communication resources has to be bounded. This property is non-iterative. It is a safety property – if the sequence is illegal (the application sends more than 100 messages) then there is exists a prefix such that any continuation of this prefix is an illegal sequence.

Property 12 combines a safety property 11 and a liveness property 6, but as a whole is neither of them. It states that at most 100 SMS messages can be sent during an application run and eventually a particular audit action has to be done (for example making a backup of the application state). This property is not a safety property because of the eventual audit action, it is also not a liveness one since if the application sent more than 100 SMS, the trace can never become valid again. It is also not an iterative property because a concatenation of two runs that sent 100 SMS each does not produce a valid trace. But this property is a renewal one for the same reason why properties 11 and 6 are renewal.

Property 13 Assume a repeating process (like a transactional one) where after every k transactions a report should be sent. In case the report is not sent on time, a letter with explanations should eventually be sent. This is a liveness property but it is not iterative because the property holds for 1 transaction and for $k-1$ transactions, but their concatenation is not valid because the report for the k transactions is not sent.

These properties above are paradigmatic of the distinction between iterative and non-iterative properties. Intuitively speaking, *iterative properties are properties in which the number of times a legal sequence is repeated does not matter.* We call this repeating sequence an *iteration.* Of course, within the iteration itself the number of times a particular action is repeated might make a difference between being legal and being illegal, but once an iteration is legal, it can be repeated as many times as one wishes.

4.3 Classification of Enforcement Mechanisms

4.3.1 A Model of Enforcement Mechanisms

Let us first introduce enforcement mechanisms as a sequence transformers $E : \Sigma^\infty \to \Sigma^\infty$. We consider a particular form of enforcement mechanism proposed by Bauer et al. and called *edit automaton* [7, 51]. Edit automata have a power of inserting and suppressing actions from the executions. As an example, they can wait until the illegal execution becomes legal again by suppressing its actions and then insert all the suppressed actions. They also can behave like a security automata [59] and simply suppress the suffix of the execution that makes it illegal.

We present our own definition of this automaton that is slightly different from the original definition in [7] and the refined definition in [51]. Intuitively, we have just simplified the original notions by enucleating the notions of output and memory and always forced the enforcement mechanism to progress in the processing of the input. We later show that our actions are identical to the combinations of atomic actions (read symbol but no output, output symbol but don't read input) from [51] on every non-diverging computation (a diverging computation is a computation where the edit automaton will run forever without reading any input while keeping outputting data). We first present our own definition of edit automaton.

Definition 4.3.1 (Edit Automata) *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 target with action set Σ. Q specifies the possible states, and $q_0 \in Q$ is the initial state. The total function $\delta : Q \times \Sigma \to Q$ specifies the transition function; the total function $\gamma_o : Q \times \Sigma^* \times \Sigma \to \Sigma^*$ defines the output according to the current state, the sequence of actions kept so far and the current input action; the total function $\gamma_k : Q \times \Sigma^* \times \Sigma \to \Sigma^*$ defines the sequence that will be kept after committing the transition.*

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

When the automaton proceeds with one more input action, the function γ_o specifies the output of the automaton at this transition and the function γ_k specifies the memory containing the actions that are processed by the automaton but not output yet. The keep function will add the input action to the memory or will ignore the input action. In general case the keep function can perform more actions on the current memory, for example it can add arbitrary actions to it.

Definition 4.3.2 (Run of Edit Automaton) *Let $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ be an edit automaton. A* run *of automaton E on an* input *sequence of actions $\sigma_{in} = a_1; a_2; \ldots$ is a sequence of pairs $\big\langle (q_0, \epsilon), (q_1, \sigma_1^k), (q_2, \sigma_2^k), \ldots \big\rangle$ such that $q_{i+1} = \delta(q_i, a_{i+1})$ and $\sigma_{i+1}^k = \gamma_k(q_i, \sigma_i^k, a_{i+1})$. The* output *of E on input σ_{in} is sequence of actions $\sigma_o = \sigma_1^o; \sigma_2^o; \ldots$ such that $\sigma_{i+1}^o = \gamma_o(q_i, \sigma_i^k, a_{i+1})$.*

We will use the following notation for a finite run of n steps as

$$(q_0, \sigma) \overset{\sigma_o}{\rightsquigarrow}_A (q_n, \sigma[n+1..])$$

which means that the automaton A with initial state q_0 on input sequence σ processes a prefix $\sigma[..i]$ of this input and finishes in state q_n while producing output σ_o.

Now we present the definition of edit automata from [51]. Every execution of an edit automaton is specified using a labeled operational semantics. The basic single-step has the form

$$(q, \sigma) \xrightarrow{\tau} (q', \sigma')$$

where q is the current state of the automaton, σ is the sequence of actions that is in the input, q' and σ' are the state and sequence of actions after the automaton takes a step, and τ is an output sequence. The definition of edit automaton verbatim from [51] is as follows:

> An *edit automaton E* is a triple (Q, q_0, δ_L) defined with respect to some system with action set Σ. As with truncation automata, Q is the possibly countably infinite set of states, and q_0 is the initial state. In contrast to truncation automata, the deterministic and total transition function δ of an edit automaton has the form $\delta_L : (Q \times \Sigma) \to Q \times (\Sigma \cup \{\cdot\})$. The transition function specifies, when given a current state and input action, a new state to enter and either an action to *insert* into the output stream (without consuming the input action) or the empty sequence to indicate that the input action should be *suppressed* (i.e., consumed from the input without being made observable).
>
> $$\frac{\sigma = a; \sigma' \quad \delta_L(q, a) = (q', a')}{(q, \sigma) \xrightarrow{a'} (q', \sigma)} \qquad \text{(E-INS)}$$
>
> $$\frac{\sigma = a; \sigma' \quad \delta_L(q, a) = (q', \cdot)}{(q, \sigma) \xrightarrow{\cdot} (q', \sigma')} \qquad \text{(E-SUP)}$$

Remark 4.3.1 *In the original paper [51] the transition function δ_L was defined as an arbitrary function of the signature mentioned above in the quotation.*

In [51] Ligatti et al. argue that this single-step semantics can easily simulate multi-step semantics. In the rest of this section we denote the edit automaton defined in [51] by the wording single step edit automaton.

This automaton allows diverging computation. Formally, it means that after some action i of some finite input, the automaton will not read any more input symbols and will only output symbols.

Definition 4.3.3 (Diverging computation) *A* diverging computation *starting in the state q_1 of the single step edit automaton (Q, q_0, δ_L) and triggered by $\sigma \in \Sigma^*$ is a sequence $\langle q_1, q_2, \ldots\rangle \in Q^\omega$, where Q^ω is an infinite sequence of states from the set Q, and $(q_i, \sigma) \xrightarrow{a'_i} (q_{i+1}, \sigma)$ for some $a'_i \in \Sigma$ for all $i \geq 1$,*

Definition 4.3.4 (Effectively diverging computation) *A single step edit automaton $E = (Q, q_0, \delta_L)$ has an* effectively diverging computation *if there exists a finite sequence $\sigma \in \Sigma^*$ and there exists a finite sequence of states $\langle q_1, q_2, \ldots, q_n\rangle$ such that*

1) $\sigma_0 = \sigma$, and

2) for all $i \leq n$

- *either $(q_i, \sigma_i) \xrightarrow{a'_i} (q_{i+1}, \sigma_i)$ for some $a'_i \in \Sigma$*
- *or $\sigma_i = a;\sigma_{i+1}$ and $(q_i, \sigma_i) \xrightarrow{\cdot} (q_{i+1}, \sigma_{i+1})$ for some $a \in \Sigma$ and $\sigma_i \in \Sigma^*$, and*

3) there exists a diverging computation starting in q_n and triggered by σ_n.

While it was theoretically useful in [51], the very idea that an enforcement mechanism could possibly produce output without any input is not acceptable by the end users from our industrial e-health case study. In contrast, the idea that the enforcement mechanism could spend a lot of time in order to process an input and eventually report a long sequence of follow-up actions was considered impractical but possible.

So our standpoint is that the only way to produce an infinite output should be to get an infinite input. Therefore, differently from the original definition in [51], our automaton consumes an input action a at every transition.

Proposition 4.3.1 *For all enforcement mechanisms without effectively diverging computations edit automata from Definition 4.3.1 and single step edit automata [51] are identical.*

Proof. We show how to construct an edit automaton from Definition 4.3.1 given an edit automaton from the original definition [51]. For all a, σ', q

1) if $(q, a; \sigma') \xrightarrow{\cdot} (q', \sigma')$ then $\delta(q, a) = q'$ and for all σ_k the output and keep functions are $\gamma_o(q, \sigma_k, a) = \cdot$ and $\gamma_k(q, \sigma_k, a) = \cdot$

2) if $(q, a; \sigma') \xrightarrow{a'} (q', a; \sigma')$ then one of the two following cases holds:

   (a) let $\langle q_1, \dots q_n \rangle \in Q^*$ be the longest sequence such that $q = q_1$ and $(q_i, a; \sigma') \xrightarrow{a'_i} (q_{i+1}, a; \sigma')$ for all $0 < i < n$ and $(q_n, a; \sigma') \xrightarrow{\cdot} (q_{n+1}, \sigma')$ then $\delta(q, a) = q_{n+1}$ and for all σ_k the output and keep functions are $\gamma_o(q, \sigma_k, a) = a'_1; \dots a'_{n-1}$ and $\gamma_k(q, \sigma_k, a) = \cdot$

   (b) let $\langle q_1, \dots q_n \rangle \in Q^*$ be the sequence such that $q = q_1$ and $(q_i, a; \sigma') \xrightarrow{a'_i} (q_{i+1}, a; \sigma')$ for all $0 < i < n$ and then the automaton stops proceeding the input and outputting, because there are no successors from the state q_n, then $\delta(q, a) = q_n$ and for all σ_k the output and keep functions are $\gamma_o(q, \sigma_k, a) = a'_1; \dots a'_{n-1}$ and $\gamma_k(q, \sigma_k, a) = \cdot$.

3) otherwise let $\delta(q, a) = q_\perp$ and for all σ_k the output and keep functions are $\gamma_o(q, \sigma_k, a) = \cdot$ and $\gamma_k(q, \sigma_k, a) = \cdot$.

It is easy to show that both automata have the same I/O relation. For example, for the case 2(b): even though the constructed automaton will proceed with the input action a, since the automaton stops executing the input, the observed behavior (outputting $a'_1; \dots a'_{n-1}$) is still the same. The only difficult part is to show that we can never reach a state $q_\perp$. Since the edit automaton has no effectively diverging computations this means that either $q_\perp$ is not reachable by a finite prefix (condition (2) of Definition 4.3.4) or there cannot be a diverging computation starting in q (condition (3) of Definition 4.3.4). So the state q_n in the construction must exist and thus $q_\perp$ is not reachable.

$\square$

In the proof of Proposition 4.3.1 the keep function γ_k always returns an empty sequence. This is because γ_k is an internal function to the automaton: it defines a new value of the suspended sequence σ_k that is not used in the proof.

4.3.2 A New Classification of Edit Automata

As we showed in Chapter 3, the same security policy can be enforced by an edit automaton in different ways. Both edit automata that we have seen in that chapter do enforce the same market policy, but they are different. We will propose a fine grained classification of edit automata according to the way they enforce security properties.

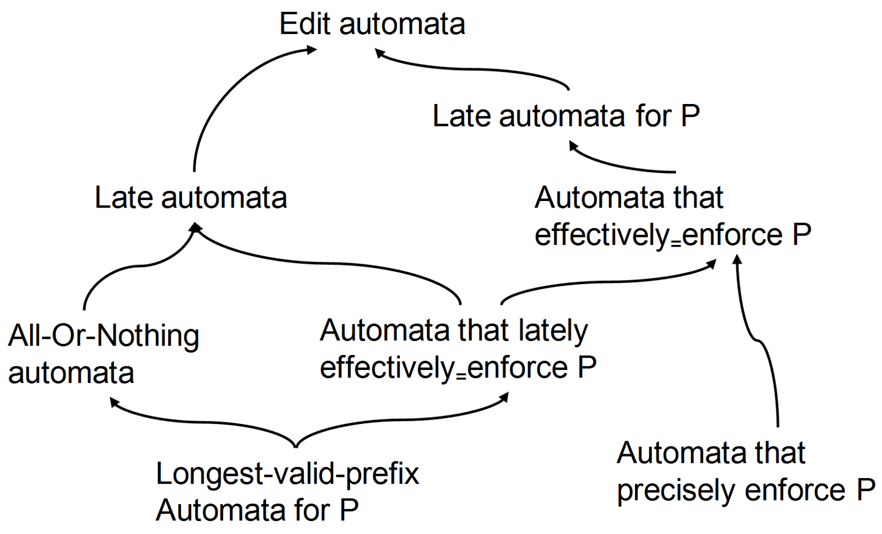

Edit automata
Late automata for P
Late automata
Automata that effectively$_=$enforce P
All-Or-Nothing automata
Automata that lately effectively$_=$enforce P
Longest-valid-prefix Automata for P
Automata that precisely enforce P

Figure 4.2: Relation between the classes of edit automata

In Figure 4.2 we anticipate the main results of this section by showing the hierarchy of different kinds of edit automata that we explain and prove later in this section. Later in Figure 4.7 we will pictorially describe the precise relations among different kinds of edit automata.

We start with a wide class of edit automata called *Late automata.* They simply output some prefix of the input. This class will be the container of other less trivial cases when the property $\widehat{P}$ will be taken into account.

Definition 4.3.5 (Late automata) *A* Late automaton A *is an edit automaton* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ *with the restriction that it always outputs some prefix of the input:*

$$\forall i \in \mathbb{N} : \exists j \leq i : \exists q' \in Q : (q_0, \sigma) \overset{\sigma[..j]}{\rightsquigarrow}_A (q', \sigma[i+1..]) \tag{4.7}$$

Property (4.7) means that for every processed prefix $\sigma[..i]$ of the input sequence there exists a run of edit automaton A such that it outputs some prefix $\sigma[..j]$ of the processed input sequence $\sigma[..i]$. We call this property *output latency* since it means that at every step of execution automaton outputs some prefix of the input.

The example of Late automaton is shown in Figure 4.3. This automaton simply outputs the first action of the input after reading the second one and then outputs second and third actions after reading the third action.

In order to give a formal definition of the automata from Theorem 8 [7] for any property $\widehat{P}$ we present also a wider class of automata called *All-Or-Nothing automata.*

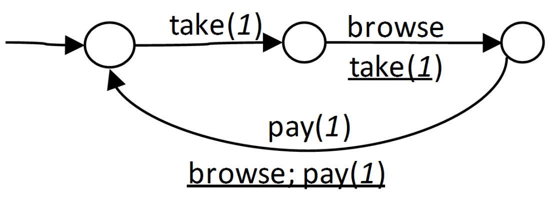


Figure 4.3: Example of Late automaton.

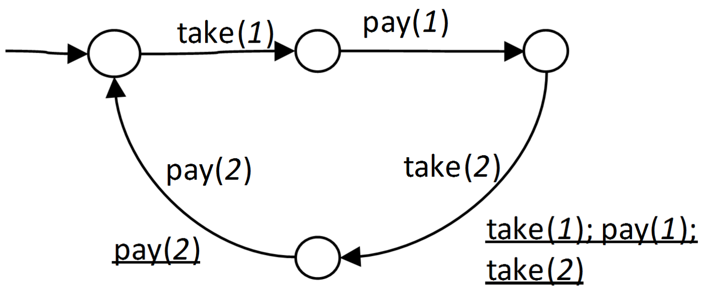


Figure 4.4: Example of All-Or-Nothing automaton.

These automata always output some prefix of the input (hence it is a particular kind of Late automata). Moreover, on every transition they either output all suspended input actions or suppress the current action.

Definition 4.3.6 (All-Or-Nothing automata) *An* All-Or-Nothing automaton A *is an edit automaton* $A = \langle Q, q_0, \delta, \ \gamma_o, \gamma_k \rangle$ *with the following restrictions:*

- *This automaton outputs a prefix of the input: property (4.7) holds.*
- *At every step of the transition either it outputs the whole suspended sequence of actions (the input symbols read by the automaton but not in the output yet) or suppresses the current action:*

$$\begin{array}{c}(\gamma_o(q, \sigma_k, a) = \sigma_k; a \ \textit{and} \ \gamma_k(q, \sigma_k, a) = \cdot) \ \textit{or} \\ (\gamma_o(q, \sigma_k, a) = \cdot \ \textit{and} \ \gamma_k(q, \sigma_k, a) = \sigma_k; a)\end{array} \tag{4.8}$$

An example of All-Or-Nothing automaton is given in Figure 4.4.

The next step is the refinement of this class towards what we call *Longest-valid-prefix* Automata for $\widehat{P}$[1]. These automata always output a prefix of the input (hence it is a particular kind of Late automata) and they are particular kind of All-Or-Nothing

[1]In the previous papers [12, 17, 14] we called it *Ligatti* automaton.

automata. Moreover, they output the longest valid prefix. The definition of Longest-valid-prefix automaton for property $\widehat{P}$ given below was made according to the construction of edit automaton given in the proof of Theorem 8 [7].

Definition 4.3.7 (Longest-valid-prefix automata for property $\widehat{P}$) *A* Longest-valid-prefix automaton E *for property $\widehat{P}$ is an edit automaton $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with the following restrictions:*

- *The automaton outputs a prefix of the input (4.7).*
- *Either it outputs the whole suspended sequence of actions or suppresses the current action (4.8).*
- *Output is valid at every transition (here σ' is an already output sequence)*

$$\widehat{P}(\sigma'; \gamma_o(q, \sigma_k, a)) \tag{4.9}$$

- *If in the state q the current sequence $\sigma'; \sigma_k; a$ is valid then it outputs the whole sequence:*

$$\begin{array}{l} \textit{If } \widehat{P}(\sigma'; \sigma_k; a) \textit{ then} \\ \gamma_o(q, \sigma_k, a) = \sigma_k; a \textit{ and } \gamma_k(q, \sigma_k, a) = \cdot. \end{array} \tag{4.10}$$

At every state a Longest-valid-prefix automaton for property $\widehat{P}$ keeps the sequence that was read till the current moment (and consists of already output sequence σ' and kept but not yet output sequence σ_k) in order to decide whether $\widehat{P}(\sigma'; \sigma_k; a)$ holds. A possible way of implementing this is $Q = \Sigma^*$. In our definition a Longest-valid-prefix automaton for property $\widehat{P}$ is obviously a particular kind of edit automaton. We will show that this statement holds in the original definition as well.

Let us now remind the constructive proof of Theorem 8 [7] and show that the edit automaton constructed following this proof is a Longest-valid-prefix Automaton for $\widehat{P}$.

The proof of Theorem 8 in [7] constructs an edit automaton as follows:

- States: $q \in \Sigma^* \times \Sigma^* \times \{+, -\}$ [the sequence of actions seen so far, the actions seen but not emitted, and $+(-)$ is used to indicate that the automaton must not (must) suppress the current action]
- The initial state $q0 = \langle \cdot, \cdot, + \rangle$.
- Consider processing the action a in state q.

(A) If $q = \langle\sigma, \tau, +\rangle$ and $\neg\widehat{P}(\sigma; a)$ then suppress a and continue in state $\langle\sigma; a, \tau; a, +\rangle$.

(B) If $q = \langle\sigma, \tau, +\rangle$ and $\widehat{P}(\sigma; a)$ then insert $\tau; a$ and continue in state $\langle\sigma; a, \cdot, -\rangle$.

(C) Otherwise, $q = \langle\sigma, \tau, -\rangle$. Suppress a and continue in state $\langle\sigma; a, \cdot, +\rangle$.

Proposition 4.3.2 *The edit automaton constructed following the proof of Theorem 8 in [7] for property $\widehat{P}$ is a Longest-valid-prefix automaton for $\widehat{P}$.*

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

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

Since at every step the output is empty or $\sigma_k; a$ then the automaton obeys the property (4.8); it always outputs prefix of the input, hence statement (4.7) holds as well. Constructed automaton outputs the sequence only if it is valid, hence statement (4.9) holds. It outputs all the suppressed actions if the sequence becomes valid, therefore statement (4.10) holds as well. Since all the conditions of Longest-valid-prefix automaton for $\widehat{P}$ are satisfied, we conclude that automaton constructed following the proof of Theorem 8 in [7] for property $\widehat{P}$ is Longest-valid-prefix automaton for $\widehat{P}$.

□

Proposition 4.3.2 also holds for the construction in the proof of Theorem 3.3 in [51].

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

- edit automata can suppress arbitrary actions from the input without inserting them later and can insert arbitrary actions in the output.
- Longest-valid-prefix 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 (in other words, it outputs the longest valid prefix of the input).

Since the automaton constructed according to the proof of Theorem 8 in [7] is a Longest-valid-prefix automaton for property $\widehat{P}$ while the automaton given in [7] (Figure 3.1) is not a Longest-valid-prefix automaton, the difference between their behaviors is clear.

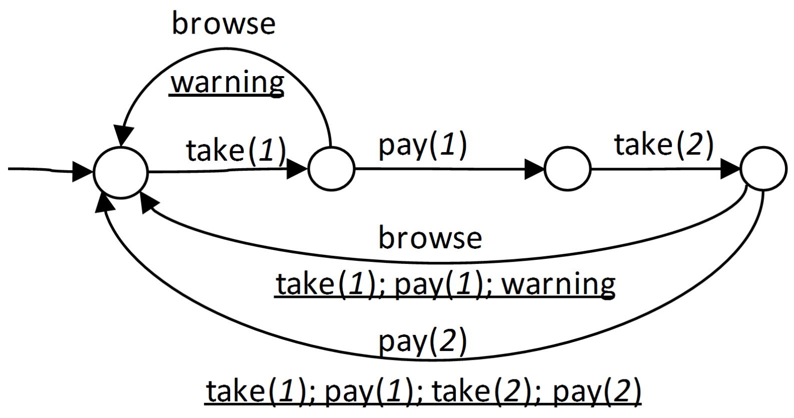


Figure 4.5: Example of Late automaton for property $\widehat{P}$.

Still, the automaton of Figure 3.1 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 4.3.8 (Late automata for property $\widehat{P}$) *A* Late automaton *A* for $\widehat{P}$ *is an edit automaton $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with the following restrictions (σ' is with the following restrictions (σ' is a sequence that is in the output already, σ_k is a sequence of an input symbols read by the automaton but not in the output yet):*
If $\widehat{P}(\sigma'; \sigma_k; a)$ then

- *Output is a prefix of the input (4.7), and*
- *Output is always valid (4.9).*

The example of Late automaton for property $\widehat{P}$ is shown in Figure 4.5. This automaton is similar to the one given in Figure 3.1 with the only difference that it delays the output of the first `take-pay` transaction.

4.3.3 Relations between the Edit Automata Classes

Many authors [7, 33, 45, 51] have noted the importance of enforcement mechanisms obeying two abstract principles, called *soundness* and *transparency*.

Definition 4.3.9 (Soundness) *The enforcement mechanism that enforces property $\widehat{P}$ is* sound *if all the outputs are legal according to the property:*

$$\forall \sigma \in \Sigma^{\infty} : \ \widehat{P}(E(\sigma)) \tag{4.11}$$

Definition 4.3.10 (Transparency) *An enforcement mechanism is* transparent *if it does not change the executions that already obey $\widehat{P}$:*

$$\forall \sigma \in \Sigma^{\infty} : \widehat{P}(\sigma) \Rightarrow E(\sigma) \approx \sigma \tag{4.12}$$

In the original papers [7, 51] notation $\tau \approx \sigma$ means that τ and σ are semantically equivalent. However, the authors did not impose any restrictions on semantic-equivalence relation except that it should be an equivalence relation (reflexive, symmetric, and transitive), and that the security property of interest does not distinguish between semantically equivalent executions. More precisely, we take a notion of Indistinguishability from [51]:

$$\forall \sigma, \sigma' \in \Sigma^{\infty} : \sigma \approx \sigma' \Rightarrow (\widehat{P}(\sigma) \iff \widehat{P}(\sigma')) \tag{4.13}$$

Since the semantic-equivalence relation is not precisely defined, we will be more interested and can be different for each target application, we will use the identity "=" relation.

One of the definition of enforcement in [7] is *precise enforcement*, that obeys both soundness and transparency.

Definition 4.3.11 (Precise Enforcement) *An automaton A with starting state q_0* precisely enforces *a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^*$ $\exists q'$ $\exists \sigma' \in \Sigma^*$ such that*

1. $(q_0, \sigma) \overset{\sigma'}{\rightsquigarrow}_A (q', \cdot)$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \forall i \in \mathbb{N} : \exists q'' \in Q : (q_0, \sigma) \overset{\sigma[..i]}{\rightsquigarrow}_A (q'', \sigma[i+1..])$

According to this 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.

There is another notion of enforcement called "effective $_=$ enforcement" [7] (with later refinement in [51]) that obeys the properties of soundness and transparency and output production (for any input there is an output).

Definition 4.3.12 (Effective$_=$Enforcement) *An automaton A with starting state q_0* effectively$_=$enforces *a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^*$ $\exists q'$ $\exists \sigma' \in \Sigma^*$ such that*

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

Then we introduce a refinement of effective$_=$enforce-ment, where an automaton can suppress some actions and later insert them when the sequence turns out to be legal. We name it *Late effective$_=$enforcement*.

Definition 4.3.13 (Late Effective$_=$Enforcement) *An edit automaton A with starting state q_0* lately effectively$_=$en-forces *a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^*\ \exists q'\ \exists \sigma' \in \Sigma^*$ such that*

1. $(q_0, \sigma) \overset{\sigma'}{\rightsquigarrow}_A (q', \cdot)$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$, *and*
4. $\forall i \in \mathbb{N} : \exists j \leq i : \exists q' \in Q : (q_0, \sigma) \overset{\sigma[..j]}{\rightsquigarrow}_A (q', \sigma[i+1..])$.

The definition above obeys four properties of enforcement: output production (1), soundness (2), transparency (3) and output latency, which is originally presented in equation (4.7).

Notice that edit automaton that lately effectively$_=$en-forces a property $\widehat{P}$ will always output some valid prefix of the input. This conclusion is obvious because soundness ensures that all output is valid and output latency ensures that output is always a prefix of the input.

We show an example of edit automaton that lately effectively$_=$enforces a property $\widehat{P}$ in Figure 4.6.

It is easy to see from the definitions that edit automata that lately effectively$_=$enforce a property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$enforce $\widehat{P}$. An example is the edit automaton in Figure 3.1 that effectively$_=$enforces property $\widehat{P}$. From an illegal input sequence `take(1)`; `browse`; `take(2)`; `pay(2)` it produces `warning`; `take(2)`; `pay(2)` while automaton in Figure 4.6 that lately effectively$_=$enforces $\widehat{P}$ outputs nothing.

As it is said in [7] edit automaton from Figure 3.1 effectively$_=$enforces the market policy (Example 3.1.1). 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.

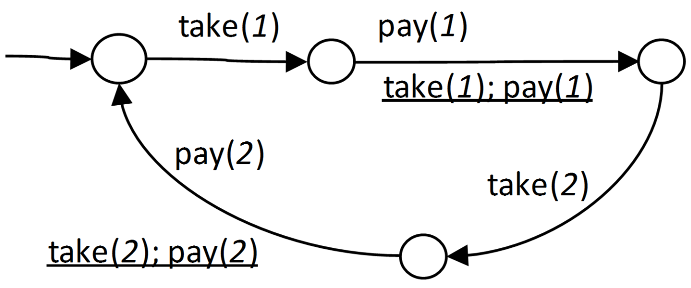


Figure 4.6: Edit automaton that lately effectively$_=$enforces property $\widehat{P}$.

Proposition 4.3.3 *Late automata and Late automata for property $\widehat{P}$ are not a proper subset of each other.*

Proof. First we prove that if an 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 (4.7). 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 sequence (4.9).

Next we 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 (4.7). □

For example, for input sequence $\sigma =$ `take(1);browse` the Late automaton from Figure 4.3 will output `take(1)` action which is not valid while the Late automaton for property $\widehat{P}$ from Figure 4.5 will output `warning` action.

From Proposition 4.3.3 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 4.3.1 *Edit automata that effectively$_=$enforce property $\widehat{P}$ are a proper subset of Late automata for $\widehat{P}$.*

Proof. First we prove that if an 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 properties $\widehat{P}(\sigma')$ and $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$. If σ is valid it outputs the whole sequence, so automaton A is a Late automaton for $\widehat{P}$ because it outputs a valid prefix (σ is a valid prefix of itself). If σ is invalid, automaton A can output an arbitrary valid sequence while Late automaton for property $\widehat{P}$ can output any arbitrary sequence (valid or invalid).

Next we have to prove that if an 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$_=$enforce-ment $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$ will not hold. □

For example, the Late automaton for $\widehat{P}$ from Figure 4.5 for a valid input `take(1)`; `pay(1)` will output nothing while the automaton from Figure 3.1 that effectively$_=$enforces property $\widehat{P}$ will output the whole input `take(1)`; `pay(1)`.

Proposition 4.3.4 *Edit automata that effectively$_=$enforce property $\widehat{P}$ are not a subset of Late automata.*

Proof. We 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 properties: $\widehat{P}(\sigma')$ and $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$. In case of invalid input, the automaton A will output some valid sequence (according to soundness), which is not necessary a prefix of the input. Therefore it is not necessarily a Late automaton. □

For example, the automaton from Figure 3.1 that effectively$_=$enforces property $\widehat{P}$ for an invalid input `take(1)`; `browse` will output the `warning` action which is not possible for a Late automaton that has to output some prefix of the input.

Theorem 4.3.2 *Edit automata that lately effectively$_=$enforce a property $\widehat{P}$ are exactly those Late automata that effectively$_=$enforce property $\widehat{P}$.*

Proof. Similarly to the definitions of late effective$_=$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 effectively$_=$enforces property $\widehat{P}$ then it obeys the property of output latency (4.7) and hence A is a Late automaton. According to definitions 4.3.12 and 4.3.13, since A lately effective-ly$_=$enforces $\widehat{P}$ it also effectively$_=$enforces $\widehat{P}$.
(*Only-if direction*). If A is a Late automaton that effectively$_=$enforces property $\widehat{P}$ then it

always outputs some valid prefix of the input (since it obeys soundness and output latency) and in case of valid input it outputs the whole input sequence (property of transparency). Hence, first three conditions of late effective$_=$ enforcement hold. The 4rd property holds as well because A is a Late automaton. Hence A lately effectively$_=$enforces $\widehat{P}$. □

It is immediate from the definitions the following results:

Proposition 4.3.5 *Edit automata that precisely enforce property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$enforce property $\widehat{P}$.*

Proof. If an edit automaton precisely enforce property $\widehat{P}$ then for the valid input sequence it will output in a lock-step mode the whole sequence, hence the property of transparency holds. All the other properties of effective$_=$enforcement hold as well.

If an edit automaton effectively$_=$enforces property $\widehat{P}$ then it is not necessary that it precisely enforces $\widehat{P}$. For an invalid input σ that has a valid prefix $\sigma' \preceq \sigma$, it can output some valid sequence which does not necessarily have a prefix σ'. □

Proposition 4.3.6 *Edit automata that precisely enforce property $\widehat{P}$ are not a subset of Late Automata.*

Proof. The proof is straightforward: for an illegal input edit automaton that precisely enforce property $\widehat{P}$ can output a sequence that is not necessarily a prefix of the input, however a Late Automaton always outputs some prefix of the input. □

Proposition 4.3.7 *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 a Late automaton. Since (4.7) holds for All-Or-Nothing automaton A, then A is a Late automaton.

Next we show that if A^* is a Late automaton then it is not necessary that A^* is an 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 an All-Or-Nothing automaton because 4.8 does not hold. □

For example, the Late automaton from Figure 4.3 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 non-empty prefix of the input sequence.

Proposition 4.3.8 *All-Or-Nothing automata are not a subset of Late automata for property $\widehat{P}$.*

Proof. We show that if A is an All-Or-Nothing automaton, then it is not necessary that A is a 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 Figure 4.4 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 4.3.9 *Edit automata that lately effectively$_=$enforces property $\widehat{P}$ and All-Or-Nothing automata are not a proper subset of each other.*

Proof. First we prove that if an edit automaton A lately effectively$_=$en-forces 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 prove that if edit automaton A^* is an All-Or-Nothing automaton then it is not necessary that A^* lately effectively$_=$enforces $\widehat{P}$. At some step A^* can output some invalid prefix of the input while automaton that lately effectively$_=$enforces $\widehat{P}$ always outputs only valid prefix of the input. □

For example, the automaton given in Figure 4.6 that lately effectively$_=$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 non-empty 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 Figure 4.4 outputs the whole input sequence, which is invalid. This automaton cannot be considered as automaton that lately effectively$_=$enforces property $\widehat{P}$ because it produces illegal output.

Theorem 4.3.3 *All-Or-Nothing automata that lately effectively$_=$enforce a property $\widehat{P}$ are exactly Longest-valid-prefix automata for property $\widehat{P}$.*

Proof. We show that All-Or-Nothing automaton A lately effectively$_=$en-forces a property $\widehat{P}$ if and only if A is a Longest-valid-prefix automaton for property $\widehat{P}$.

(*If Direction*). If automaton A is All-Or-Nothing automaton then equations (4.7) and (4.8) hold. Since A lately effectively$_=$enforces property $\widehat{P}$ then it obeys the properties of soundness (condition 2 of Definition 4.3.13) and transparency (condition 3 of Definition 4.3.13). Notice that soundness means that the output is always valid, which is exactly what equation (4.9) describes; transparency means that if input is valid then output is equal to the input, which is shown in equation (4.10). Hence, all the conditions of Longest-valid-prefix automaton for $\widehat{P}$ are satisfied.
(*Only-if direction*). If A is a Longest-valid-prefix automaton for property $\widehat{P}$ then

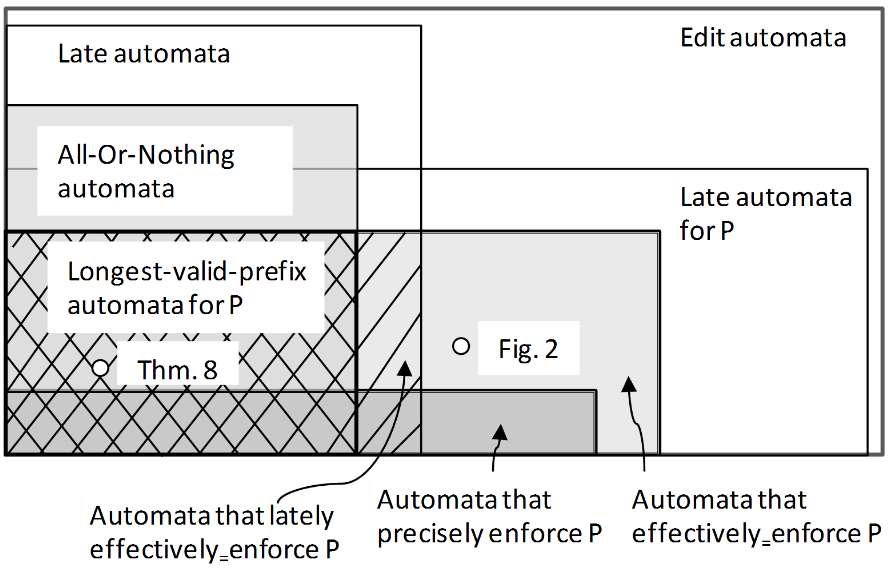
- It obeys properties (4.7) and (4.8) hence A is an All-Or-Nothing automaton, and then according to Proposition 4.3.7 it is also a Late automaton;
- It obeys properties (4.9) and (4.10) and as we noticed, they mean soundness and transparency which correspond to the conditions 2 and 3 of Definition 4.3.12. Hence, A effectively$_=$enforces $\widehat{P}$.

Therefore since A is Late automaton that effectively$_=$enforces $\widehat{P}$ then it lately effectively$_=$enforces $\widehat{P}$ according to Theorem 4.3.2 and it is an All-Or-Nothing automaton. □

4.3.4 Discussion

In this Section we have shown that the difference between the running example from Figure 2 of [7] and the edit automata that are constructed according to Theorem 8 of the same paper is due to a deeper theoretical difference. In Figure 4.7 we summarize the relations among the different kinds of edit automata that we have introduced. When drawing two boxes separated by a space we mean that inclusion is probably not proper. We have proven the correctness of this classification earlier in this section.

Now we clarify which type of edit automaton is constructed following the proof of Theorem 8 in [7] for property $\widehat{P}$ and which type of edit automaton is the one in [7] (Figure 3.1). As the Proposition 4.3.2 states, the edit automaton constructed following the proof of Theorem 8 in [7] for property $\widehat{P}$ is a Longest-valid-prefix automaton for $\widehat{P}$. The edit automaton given in Figure 3.1 [7] is an edit automaton that effectively$_=$enforces $\widehat{P}$:


Late automata
Edit automata
All-Or-Nothing automata
Late automata for P
Longest-valid-prefix automata for P
Fig. 2
Thm. 8
Automata that lately effectively_=enforce P
Automata that precisely enforce P
Automata that effectively_=enforce P

A new classification of edit automata. Here point "Thm. 8" represents an edit automaton constructed following the proof of Theorem 8 [7] and point "Fig. 2" represents an edit automaton from Figure 2 originally from [7]. We can see that both of them effectively$_=$enforce the given property, but the Longest-valid-prefix automaton from Theorem 8 is just a particular type of edit automata.

Figure 4.7: The classes of edit automata.

it obeys soundness (the automaton always outputs the valid sequence) and transparency (in case of valid input it always outputs all the sequence). The edit automaton given in Figure 3.1 [7] is not a Late automaton because it does not always output some prefix of the input (see examples 8 and 10 in Table 3.2)

Therefore we can conclude that both automata from Theorem 8 [7] and from Figure 3.1 [7] are edit automata that effectively$_=$enforce property $\widehat{P}$. But when one wants to construct such an automaton and follows the proof of Theorem 8 [7], he obtains a Longest-valid-prefix automaton for $\widehat{P}$ that lately effectively$_=$enforces $\widehat{P}$.

In other words the intuition behind the classification is the following one: Late automata always output some prefix of the input while Late automata for P outputs only valid prefixes. It is maybe surprising, but Late automata for P are not a proper subset of Late Automata precisely because of their behavior in case of non-compliant input (See Proposition 4.3.3).

The whole characterization of the automata in this classification is based on the idea that some of them depend on the policy P and hence, the behavior of the automata in the class is the same only for compliant inputs. In case of non-compliant inputs the automata only convert the inputs to compliant ones. This is exactly the case for the automaton from Figure 2 in [7]: it is a Late automaton that effectively$_=$enforces P.

An edit automaton constructed following the proof of Theorem 8 [7] is the Longest-valid-prefix automata for P, that modifies non-compliant execution in a particular way because it can only suppress the invalid input and wait until it becomes valid in the future.

4.4 Summary

In Section 4.2 have proposed a new notion of iterative security property and defined relations between the security properties from the state of the art (safety, liveness and renewal) and iterative property. The result of this contribution is presented in Figure 4.1.

We have presented a classification of particular types of edit automata in Section 4.3. From this classification and examples of edit automata, we can conclude that the main difference between different edit automata effectively$_=$enforcing the same policy is in the way they convert non-compliant sequences into compliant ones. We have proposed a solution to Issue 1, but leads us to Issue 2.

Chapter 5

Construction of Enforcement Mechanisms

As we have seen, an edit automaton constructed following the proof of Theorem 8 [7] is actually a Longest-valid-prefix automaton, a specific kind of edit automaton. In this chapter we will show how given a security property we can build a Longest-valid-prefix automaton. Later, we find out how an iterative security property can be enforced by another particular kind of edit automaton and show how to construct it.

5.1 Additional Notations

First we need to present some additional notations we will use in all the constructions of this and later chapters. It is simpler to represent the memory of the edit automaton rather than defining the keep function. The notation (q, σ_i, σ_k) represents the current configuration of edit automaton, where q is the current state, σ_i is the current input sequence, and σ_k is the current memory containing suspended actions. A transition of an edit automaton is denoted by

$$(q, \sigma_i, \sigma_k) \overset{\sigma_o}{\hookrightarrow} (q', \sigma_i', \sigma_k') \tag{5.1}$$

where σ_o is the output produced, and respectively q', σ_i' and σ_k' are the new state, the input sequence left to read and the updated memory. Notice that since we do not allow diverging computations, it is always that case that $\exists a \in \Sigma : \sigma_i = a; \sigma_i'$.

In the sequel we will show the graphical representation of an automaton. Every transition of the form: $(q, a; \sigma_i', \sigma_k) \overset{\sigma_o}{\hookrightarrow} (q', \sigma_i', \sigma_k')$ we will represent as it is shown in Figure 5.1.

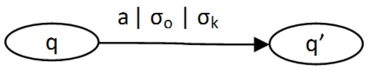


Figure 5.1: Representation of a transition of edit automaton.

We will also use the following notation. Whenever $\sigma_o = \sigma_k; a$, we will show it graphically as $*; a$ representing the current memory by $*$. Similarly, whenever $\sigma'_k = \sigma_k; a$ we will show it as $*; a$.

Before proposing the constructive procedure for edit automata we first need to formally define an input to that procedure, hence we now specify how the security property is represented in our framework.

5.2 Property Representation

In practice desired behavior of an application is often described as a workflow. We represent workflows and security properties by a new kind of finite state automaton. In our model we assume both finite and infinite execution sequences, hence we need another notion of automaton that can represent a "default" workflow.

Definition 5.2.1 (Policy automaton) *A* Policy automaton *is a tuple $\langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ is a finite nonempty set of security-relevant 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. In the following, for $q, q' \in Q$, $a \in \Sigma$ we will write $q \xrightarrow{a} q'$ whenever $\delta(q, a) = q'$. $F \subseteq Q$ is a set of accepting states.*

Definition 5.2.2 (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 $q_i \xrightarrow{a_i} q_{i+1}$. 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 5.2.3 (Property represented as Policy automaton) *Some property $\widehat{P}$ is represented as a Policy automaton A if and only if:*

$$\forall \sigma \in \Sigma^\infty : \widehat{P}(\sigma) \Longleftrightarrow A \text{ accepts } \sigma \tag{5.2}$$

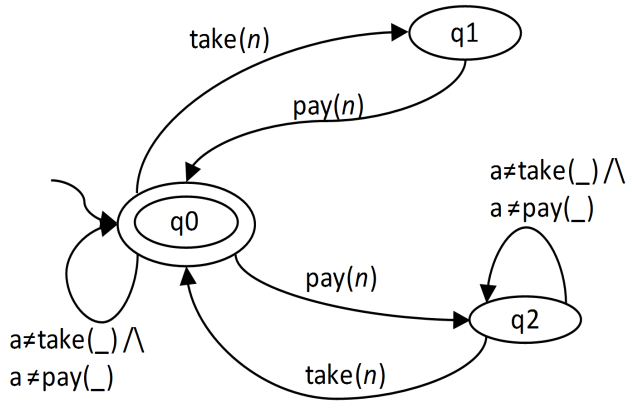


Figure 5.2: Policy automaton for a market policy.

Remark 5.2.1 *We assume that an empty sequence that corresponds to no execution of the target is always accepted by the security property ($\widehat{P}(\cdot)$ holds) and hence the initial state of the Policy automaton is always accepting: $q_0 \in F$.*

Notice that the set of infinite traces accepted by a Policy automaton represents a renewal property since a valid infinite trace corresponds to a run that goes infinitely often through an accepting state, and hence this trace has an infinite number of valid prefixes. More formally:

Proposition 5.2.1 *The set of infinite 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 a definition of renewal property (4.5).

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 $s = \langle s_0, s_1, \ldots, s_d, \ldots, s_l\rangle$ 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. □

In Figure 5.2 we present a Policy automaton for the market policy from Example 3.1.1. We have noticed that the text of the example leaves open a number of interpretations.

Remark 5.2.2 *Notice that the set of security relevant actions Σ is finite, and a number of apples in the market, that one can take or pay for, is finite as well. Hence, the Policy automaton in Figure 5.2 has finite number of transitions.*

It is clear that good sequences must have a pair of `take(n)` and `pay(n)` as the text implies, so the automaton accepts these sequences. Otherwise if after `take(n)` action there were some action different from `pay(n)` then the policy would be violated and the automaton would halt. If after `pay(n)` action there are some other actions different from `pay` and `take` (like `browse`) then the automaton simply waits for the `take(n)` action. It is not clear whether another `take(m)` action is allowed by example at this point, so that the resulting sequence of actions can be `pay(n)`; `take(m)`; `pay(m)`; `take(n)`. The text seems to imply that this is not possible, so the automaton will halt whenever `take(m)` appears after `pay(n)` and $m \neq n$.

Similarly, in Figure 5.3 we present an automaton corresponding to the BPMN description of the drug selection subprocess (Figure 3.4). This example precisely defines all the possible valid sequences of actions and is easily represented as an automaton.

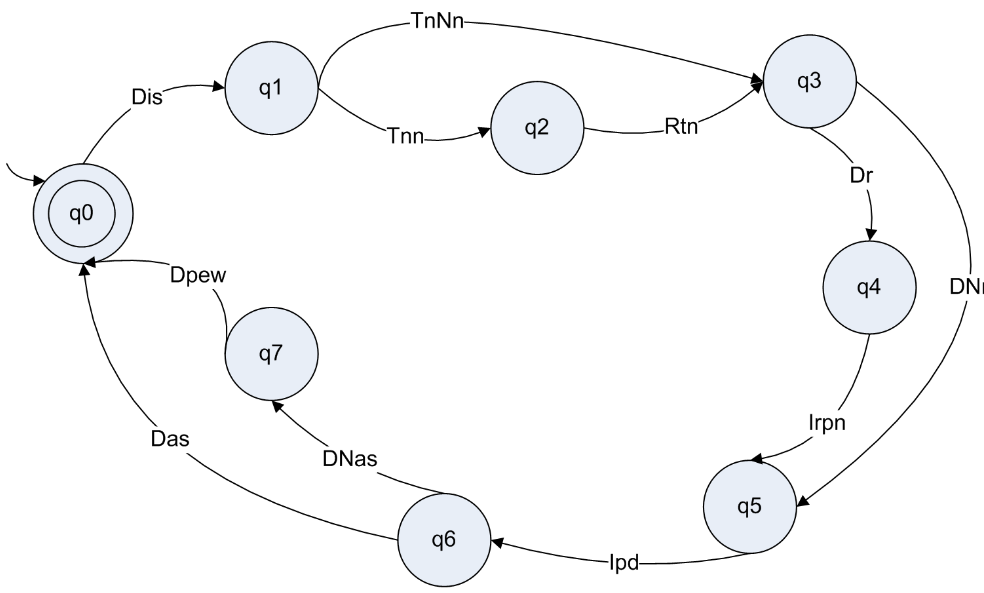
5.3 Longest-valid-prefix Automata

We have seen in Chapter 4 the classification of edit automata and the Longest-valid-prefix automata in particular. This automata are constructed from the security policy following the proof of Theorem 8 of [7], and, as we have defined, they always output the longest valid prefix of the execution.

5.3.1 Construction

According to Definition 5.2.1, the security policy is represented by a finite-state Policy automaton $A^P = \left\langle \Sigma, Q^P, q_0^P, \delta^P, F^P \right\rangle$. We construct the Longest valid prefix automaton in Figure 5.4.

Let us describe the idea behind this construction. Suppose the current state of the automaton is q, the next incoming action is a. If there is a transition in the Policy automaton from the state q on an action a to an accepting state, then the input read so far is accepted by the Policy Automaton. Therefore, we output all the actions read so far followed by the current action (we output $\sigma_k; a$ in rule [Good-Out]). If the next state is non-accepting, it means that possibly there is a path to the accepting state, so the sequence read so far can become good. Therefore we simply keep the current action in the memory (we put in the memory $\sigma_k; a$ and output nothing in rule [Good-Wait]).



Abbreviations

Dis = Drug is selected	DNr = Drug is Not for research
Tnn = Therapeutical notes needed	Ipd = Insert prescription details
Rtn = Review therapeutical notes	DNas = Drug is Not available in stock
TnNn = Therapeutical notes Not needed	Dpew = Drug physically exists in the ward
Dr = Drug is for research	Das = Drug is available in stock
Irpn = Insert research protocol number	

Figure 5.3: Policy automaton for a drug selection subprocess.

If there is no transition from state q on action a, it means that there is no path to some accepting state of the Policy automaton, and the sequence can never become valid again. So the next state in the automata has to be a new error state $q_{\perp}$, the Longest-valid-prefix automaton will output nothing but keep all the input, and this corresponds to the rule [ERROR$_{\text{KEEP}}$].

The behavior of the constructed Longest-valid-prefix automaton is exactly the same as of one constructed by the proof of Theorem 8 of [7]: it always outputs the longest valid prefix of the input. The only difference is that in the proof Theorem 8 the state of automaton contains all the read actions and if the trace can never become good again there will be as many states as the length of the trace. In our construction, as soon as the trace cannot become good again, the next state will be an error state and all following input actions will be kept. To optimize the construction we can also skip all the following inputs once the automaton is in the error state. Then the rule [ERROR] can be rewritten

$$\textsc{Good-Out}\ \frac{q \xrightarrow{a} q' \qquad q' \in F^P}{(q, a; \sigma_i, \sigma_k) \overset{\sigma_k;a}{\hookrightarrow} (q', \sigma_i, \cdot)}$$

$$\textsc{Good-Wait}\ \frac{q \xrightarrow{a} q' \qquad q' \notin F^P}{(q, a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} (q', \sigma_i, \sigma_k; a)}$$

$$\textsc{Error}_{\text{KEEP}}\ \frac{Otherwise}{(q, a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} (q_\perp, \sigma_i, \sigma_k; a)}$$

Figure 5.4: Semantics for Longest-valid-prefix automaton constructed from the Policy automaton.

as follows:

$$\textsc{Error}\ \frac{Otherwise}{(q, a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} (q_\perp, \sigma_i, \cdot)}$$

In the sequel an enforcement mechanism that outputs the longest valid prefix will be called E_{LP}, where LP stands for "the longest prefix".

5.3.2 Formal Properties

According to our construction, the resulting Longest-valid-prefix automaton for security property $\widehat{P}$ is sound sine all it's outputs are valid prefixes, and it is transparent since the longest valid prefix of a valid input is this input.

Proposition 5.3.1 *A Longest-valid-prefix automaton with the semantics from Figure 5.4 for a renewal property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$. This automaton always outputs the longest valid prefix of the input.*

Proof. Given a Policy automaton A^P we construct a Longest-valid-prefix automaton E following according to the semantics from Figure 5.4. Let us show that automaton E always outputs the longest valid prefix of the input according to the property $\widehat{P}$.

Let us denote the state of E after executing sequence σ with q, and a is the next incoming action. Let us show that this automaton maintains the invariant $\textsc{Inv}_L(\sigma)$ that

σ is the input seen so far and σ_o has been output, where σ_o is the longest valid prefix of σ according to $\widehat{P}$. Initially $\textsc{Inv}_L(\cdot)$ holds because $\widehat{P}(\cdot)$ and automaton has not output anything so far. Let us assume that $\textsc{Inv}_L(\sigma)$ holds and prove that $\textsc{Inv}_L(\sigma; a)$ holds as well in any action a.

1) If $\widehat{P}(\sigma; a)$, then there is an accepting run of A^P on $\sigma; a$. It corresponds to the rule [Good-Wait]. In this case the output of E is $\sigma_k; a$. The memory was obtained in a following way. Every time when the non-accepting state is reached, the action is kept in the memory (rule [Good-Wait]) and when finally the sequence becomes valid, a corresponding state of A^P is accepting and the memory of E contains all the read actions. Hence, the output in this case will be $\sigma; a$ and $\textsc{Inv}_L(\sigma; a)$ holds.

2) If $\sigma; a$ is not irremediable[1], then it may become legal again in the future. It means that while executing σ the path in E will correspond to the path in A^P, to be more precise, for every transition $q \xrightarrow{\sigma[i]} q'$ of the policy automaton, there will be a transition $(q, \sigma[i, ..], \sigma_k) \overset{\tau_o}{\hookrightarrow} (q', \sigma[i+1, ..], \tau_k)$, where τ_o, τ_k are either a pair $\cdot, \sigma_k; \sigma[i]$ or a pair $\sigma_k; \sigma[i], \cdot$. Therefore, only statements premises of rules [Good-Out] and [Good-Wait] will be satisfied. Hence E outputs the longest valid prefix and the memory contains all the not yet output actions that make a trace bad and $\textsc{Inv}_L(\sigma; a)$ holds.

3) If $\sigma; a$ is irremediable then there was an action $\sigma[j]$ (or a) in the input such that there was no transition in A^P from the state where the run $\sigma[..j-1]$ (or σ) arrived on the input action $\sigma[j]$. At that point a premise for rule [Error] was satisfied, after which no output can be produced. Therefore, if there exists a valid prefix, then it was output when the corresponding accepting state of A^P was reached before the sequence became irremediable. It means that E outputs the longest valid prefix and $\textsc{Inv}_L(\sigma; a)$ holds.

Therefore, in all the cases the invariant $\textsc{Inv}_L(\sigma; a)$ is maintained and hence the Longest-valid-prefix automaton E with the semantics from Figure 5.4 always outputs the longest valid prefix. Therefore, in case of valid input the automaton will not change the input (transparency maintained) and in case of invalid input it will output a valid trace (soundness maintained). □

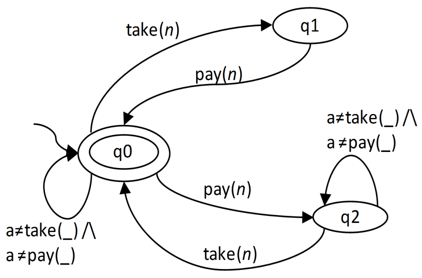


Figure 5.5: Policy automaton for market policy

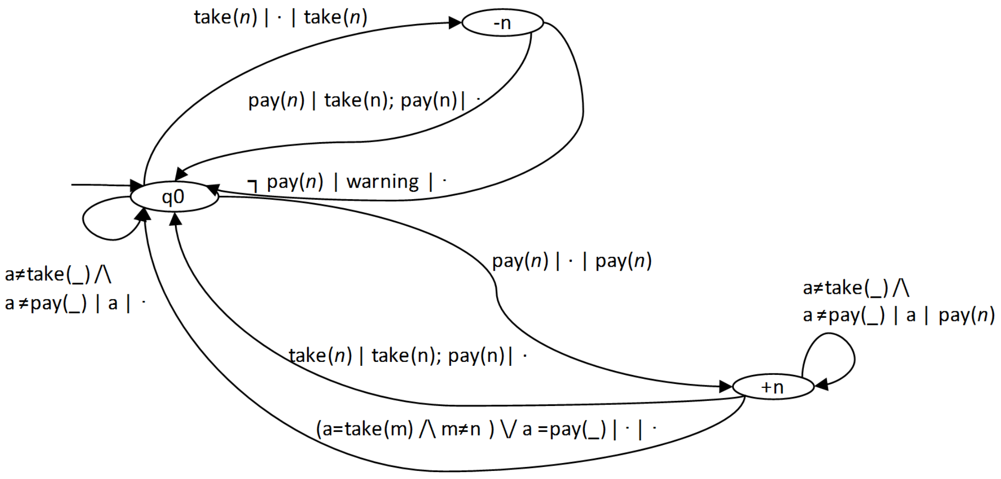


Figure 5.6: Edit automaton for market policy verbatim from [7]

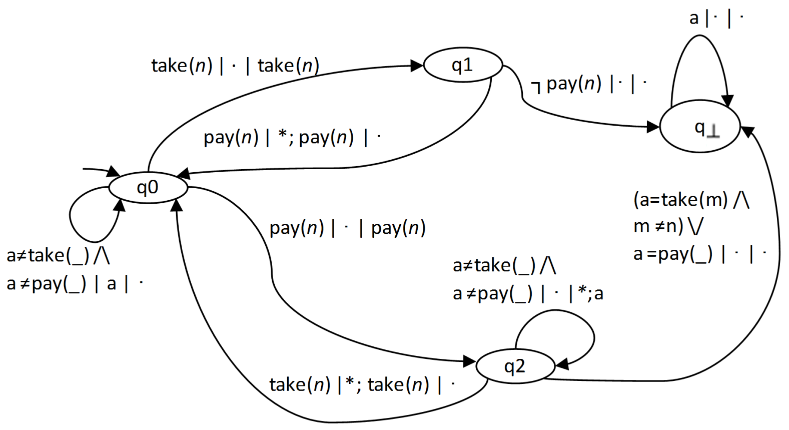


Figure 5.7: Longest-valid-prefix automaton constructed for market policy

5.3.3 Enforcement of the Market Policy

We remind the representation of the market policy as a Policy automaton in Figure 5.5, the Longest-valid-prefix automaton with the semantics from Figure 5.4 in Figure 5.7, and the original edit automaton from [7] is in Figure 5.6. To ease the comparison we represent the original edit automaton with our new notation.

It is easy to see that these automata enforce the same policy in different ways. This is due to the fact that the Longest-valid-prefix automaton from Figure 5.7 always outputs the longest valid prefix of the execution sequence while the edit automaton from Figure 5.6 enforces the given property in some other way. Even if we look in more detail at legal sequences, we already can see the difference. The legal sequence `pay(n)`; `browse`; `take(n)` is not transformed by the Longest-valid-prefix automaton (transparency holds), but the edit automaton from [7] transforms it to `browse`; `take(n)`; `pay(n)`. In our opinion, it is more desirable to leave the execution sequence without changes when it is valid.

Let us compare the transformation of illegal sequences, for example `pay(n)`; `take(m)`; `take(m)`; `pay(m)`. The Longest-valid-prefix automaton from Figure 5.7 will not produce any output because this sequence does not have a valid prefix. On the other hand, the edit automaton from Figure 5.6 will produce some output: `take(m)`; `pay(m)`. We can see that this automaton has a power of "restarting" (it is coming back to the initial state as if no error occured) and that is why in some cases it can produce more output.

5.4 Iterative Suppression Automata

For iterative properties that are represented by a Policy automaton we propose a better enforcement than outputting the longest valid prefix. We call it *Iterative Suppression.* Since the property enforced is iterative, it describes good traces that consist of independent parts, called "iterations". The idea is that this enforcement mechanism is able to recognize when a new good iteration can start, so it can suppress all the bad actions that happen between the good iterations of a tentative execution.

5.4.1 Construction

A new semantics shown in Figure 5.8 is obtained from the previous construction (Figure 5.4) by changing the condition for the traces that cannot become good again, to be

[1]As we have defined on page 42, it is a sequence of actions such that there is not suffix that can make it legal again.

more precise we add new rules [Iteration-Out and Iteration-Wait] that can recognize a beginning of a new iteration.

We start with the Policy automaton $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ representing the security property. The states of Iterative Suppression automaton are composed from two states of the Policy automaton: $Q = \{(q, q_F) | q \in Q^P \cup \{q_\perp\}, q_F \in F^P\}$, where $q = q_F$ if $q \in F^P$, or $q \neq q_F$ if there exists a run σ such that q_F is the last accepting state before reaching q. According to our assumption (see Remark 5.2.1), the initial state of the Policy automaton is accepting, hence the initial state of the Iterative Suppression automaton is (q_0, q_0).

We propose this construction of the state because an edit automaton has to "remember" the last accepting state visited during the run and compare the tentative execution to the new iterations starting only from that last visited accepting state.

Premises of the rules [Iteration-Out and Iteration-Wait] correspond to the case when the next action a is not an action recognized by the Policy automaton but this action can start a new iteration from the last visited accepting state q_F. Then, if transition on (q_F, a) brings to an accepting state q' of the Policy automaton, then a is immediately output and the memory is empty; next state is (q', q'). If next incoming action a is not accepted, then the memory is cleaned and only a is added to the memory. If a is not starting a new iteration (rule [Error]), then there is a transition to an error state and action a is not kept in the memory.

The main difference with Longest-valid-prefix automaton is that our automaton is able to recognize new good iterations and suppress only actions that caused violation of the property. We will show how the Longest-valid-prefix automaton enforces a market policy from Section 3.1 and a drug selection process from the case study of Section 3.2.

In our previous work [17] we required that all good iterations must have a *unique starting action* – an action that never repeats again in the iteration. We have then explained in our follow-up work [15] that even if there is no unique starting action, the iterative suppression automaton still soundly and transparently enforces iterative properties. If an action a can start a new iteration and also appears in the middle of another iteration, our construction will first build the transitions that repeat the Policy automaton (rules [Good-Out] and [Good-Wait]) and for the rest of the cases will check whether a can start a new iteration.

Even though it is not necessary to have unique starting actions, we think it is interesting to compare our mechanism for the properties with and without it. In both cases the mechanism is sound and transparent (as we prove below), but the modification of the invalid sequences will be done somewhat differently. Imagine a property where all the good traces match the pattern $(a; b; a; c)^*$. Now consider a tentative execution $a; b; (a; b; a; c)$.

$$\textsc{Good-Out}\ \frac{q \xrightarrow{a} q' \qquad q' \in F^P}{((q, q_F), a; \sigma_i; \sigma_k) \overset{\sigma_k;a}{\hookrightarrow} ((q', q'), \sigma_i, \cdot)}$$

$$\textsc{Good-Wait}\ \frac{q \xrightarrow{a} q' \qquad q' \notin F^P}{((q, q_F)a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F), \sigma_i, \sigma_k; a)}$$

$$\textsc{Error}\ \frac{Otherwise}{((q, q_F), a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q_\perp, q_F), \sigma_i, \cdot)}$$

$$\textsc{Iteration-Out}\ \frac{q \xrightarrow{a} \perp \qquad q_F \xrightarrow{a} q' \qquad q' \in F^P}{((q, q_F), a; \sigma_i, \sigma_k) \overset{a}{\hookrightarrow} ((q', q'), \sigma_i, \cdot)}$$

$$\textsc{Iteration-Wait}\ \frac{q \xrightarrow{a} \perp \qquad q_F \xrightarrow{a} q' \qquad q' \notin F^P}{((q, q_F), a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F), \sigma_i, a)}$$

The semantic rules in the upper part of the figure are those from the Longest-valid-prefix automaton construction and we keep them without changes. Two rules in the lower part of the figure represent recognition of iterations. Here and in the next similar figures the [Error] rule should be the last one in the list of rules, but we keep it in the upper part of the figure to underline the similarity with other constructions in this thesis.

Figure 5.8: Semantics for Iterative Suppression automaton constructed from the Policy automaton.

After processing $a; b; a$, on the next input action b our mechanism will make a transition to the error state (since it is expecting c to arrive) and exit this state only on the third action a. So the output will be empty. If the pattern did not have the second a inside (for example, $a; b; d; c$) and an execution would be $a; b; (a; b; d; c)$ then our mechanism would recognize that the second a actually starts a new iteration, so the output will be $a; b; d; c$.

However, we did not find in practice the repetition of initial actions. This is somewhat obvious: different execution patterns corresponds to execution of different macro-processes in real life and thus different starting points: starting an HIV drug dispensation process to an outpatient or a transplant process are different and the eventual dispensation of a drug in a transplant process to an inpatient is intrinsically a different action for our stakeholders.

Interestingly, this observation is not always true for other actions in the process. The doctor can repeat some actions because a process in the medical organization, may contain different sub-processes that can be repeated. Even if this observation is valid in some particular cases of particular processes, we do not discuss it later in the thesis because we do not want an enforcement mechanism to find a first half of an iteration A separately from the last part of an iteration B and concatenate them. This concatenation can bring liability if, for example, the first half of one iteration A was not finished and done by a nurse, and another iteration B was not done correctly in the first half of B, but was complaint in its second half and B was done by a doctor.

5.4.2 Formal Properties

We now prove the properties of the enforcement mechanism that we propose to construct following the semantics from Figure 5.8.

Proposition 5.4.1 *An Iterative Suppression automaton with semantics from Figure 5.8 for an iterative property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$.*

Proof. Given a Policy automaton A^P let us construct an edit automaton E with semantics from Figure 5.8. Now let us show that automaton E satisfies soundness and transparency according to property $\widehat{P}$.

Let us assume that state q is a state of the automaton A^P after executing sequence σ, and a is the next incoming action. Let us assume that invariant $\textsc{Inv}_e(\sigma)$ stating that $\widehat{P}(E(\sigma))$ and if $\widehat{P}(\sigma)$ then $E(\sigma) = \sigma$ holds and prove that $\textsc{Inv}_e(\sigma; a)$ holds for any action a.

Initially $\textsc{Inv}_e(\cdot)$ holds because automaton has not output anything so far and $\cdot$ is valid. Let us assume that $\textsc{Inv}_e(\sigma)$ holds and prove that $\textsc{Inv}_e(\sigma; a)$ holds as well in any possible case.

1) If $\widehat{P}(\sigma; a)$, then there is an accepting run of A^P on $\sigma; a$. In this case the output of E is $\sigma_k; a$ (rule [Good-Out]). The memory σ_k was obtained in a following way. Every time when the non-accepting state is reached, the action is kept in the memory (rule [Good-Wait]), and when finally the sequence becomes valid, a corresponding state of A^P is accepting and is reached thus the memory of E contains all the read actions. Hence, the output in this case will be $\sigma; a$ and $\textsc{Inv}_e(\sigma; a)$ holds.

2) If $\sigma; a$ is invalid but not irremediable then while executing σ the path in E will correspond to the path in A^P, to be more precise, for every transition $q \xrightarrow{\sigma[i]} q'$ of the policy automaton, there will be a transition $(q, \sigma[i, ..], \sigma_k) \overset{\tau_o}{\hookrightarrow} (q', \sigma[i+1, ..], \tau_k)$, where τ_o, τ_k are either a pair $\cdot, \sigma_k; \sigma[i]$ or a pair $\sigma_k; \sigma[i], \cdot$. Therefore, only rules [Good-Wait] and [Good-Out] will be used. By doing so E will output the longest valid prefix and memory of E will contain all the not yet output actions that make a trace bad. Hence $\textsc{Inv}_e(\sigma; a)$ holds.

3) If $\sigma; a$ is irremediable then let $E(\sigma; a) = \sigma_o$. According to the semantics, E outputs a sequence of actions $\sigma_k; a$ only when a corresponding state of the Policy automaton is reached (rule Good-Out), otherwise it does not produce any output. Since property $\widehat{P}$ is iterative, the good sequences consist of repeated iterations, and in our construction the only way to produce an output, is to keep in the memory all the actions that are accepted by an iteration in the Policy automaton (that starts in the initial state). This means that an output σ_o consists of valid iterations, hence, $\textsc{Inv}_e(\sigma; a)$ holds.

Therefore, in all the cases the invariant $\textsc{Inv}_e(\sigma; a)$ is maintained and hence the edit automaton E with semantics from Figure 5.8 satisfies soundness and transparency. $\square$

5.4.3 Enforcement of the Market Policy

Let us show the difference in the enforcement done by the Longest-valid-prefix automaton and the Iterative Suppression automaton for market policy. We have presented the policy automaton for the market policy from Section 3.1 in Figure 5.2 and the Longest-valid-prefix automaton in Figure 5.7. Now we will build an Iterative Suppression automaton for this policy following the semantics from Figure 5.8.

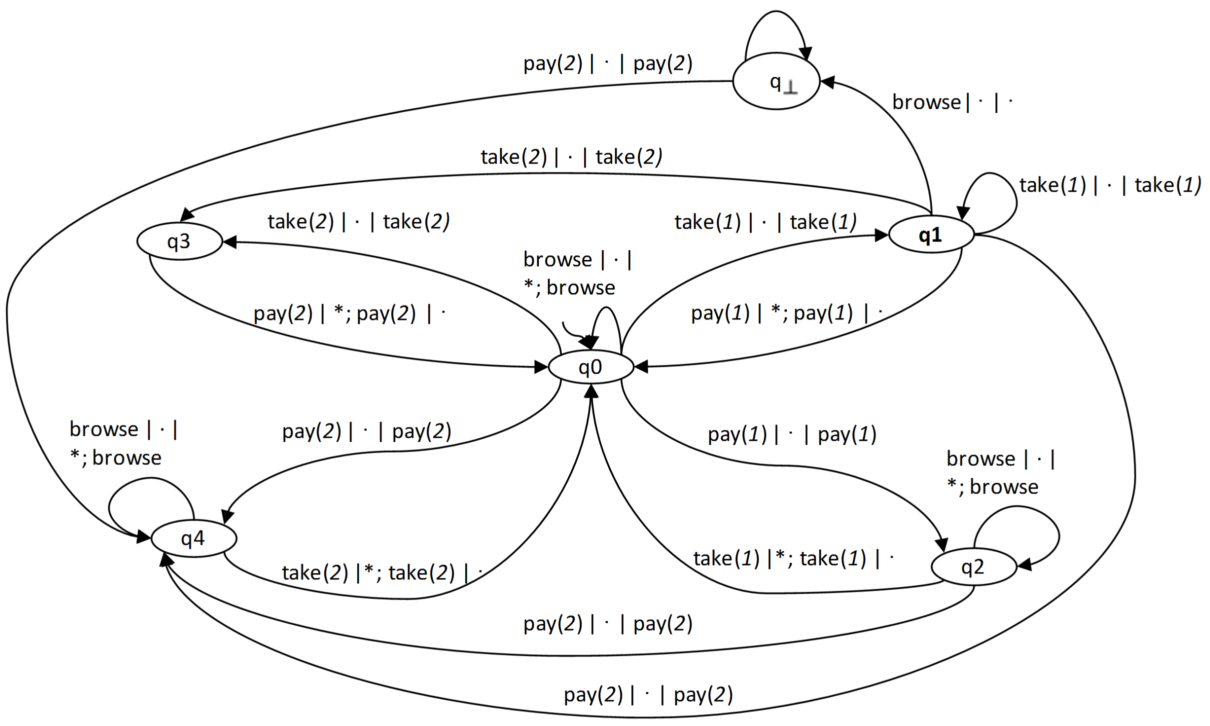


Figure 5.9: Resulting Iterative Suppression automaton for the Policy automaton for market policy.

The resulting edit automaton is shown in Figure 5.9. For the sake of readability we show an Iterative Suppression automaton for the subset of the original set of actions: {`take(1)`, `pay(1)`, `take(2)`, `pay(2)`, `browse`}. We demonstrate all the outgoing transitions only from the state $q1$ and only some of the transitions from other states for the same reason.

Now we can see that the sequences 4, 6, 8, 9, and 10 from the Section 3.1 can be transformed in a more desirable way. In Table 5.1 we show the difference in the output of Longest-valid-prefix automaton that we have constructed from the previous section and the Iterative Suppression automaton.

We can see that the output produced by Iterative Suppression automaton is more desirable since whenever the amount of paid items and taken items is the same, this part of the execution is output. Both automata obey the properties of soundness and transparency. This example shows again that soundness and transparency are not sufficient to distinguish between the execution sequence transformer, as we have stated in Issue 3.

Table 5.1: Difference in enforcement of market policy by Longest-valid-prefix automaton and Iterative Suppression automaton

No	Input	Output	
		Longest-valid-prefix automaton from Figure 5.7	Iterative Suppression automaton from Figure 5.9
4	take(1); pay(1)	take(1); pay(1)	take(1); pay(1)
6	take(1); browse; pay(2)	·	·
8	take(1); browse; pay(2); take(2)	·	pay(2); take(2)
9	take(1); pay(2); take(2)	·	pay(2); take(2)
10	pay(1); browse; pay(2); take(2)	·	pay(2); take(2)

5.4.4 Enforcement of the Drug Selection Process

In Figure 5.3 we have shown the policy automaton for the drug selection process. We will use it in the construction of Iterative Suppression automaton for this example.

We will show the difference between the enforcement by a Longest-valid-prefix automaton and Iterative Suppression automaton. We have constructed a Longest-valid-prefix automaton with the semantics from Figure 5.4. It is partially shown in Figure 5.10 (we show all transitions from the states $q_4, q_6, q_\perp$). It is easy to see that this automaton outputs the longest valid prefix: as soon as some wrong action happens (e.g. after defined that drug is for research Dr, no research protocol number is inserted !Irpn in state q_4) the automaton leads to an error state and there are no outcoming transitions from that state.

While running the drug selection process, a tentative execution consists of 3 iterations, one per each drug.

1. The first iteration is Dis; TnNn; Dr; Irpn; Ipd; Das, which is legal. We denote this iteration by ResearchRun.

2. The second iteration is Dis; TnNn; Dr; Ipd; Das, which is illegal. It means that the drug is submitted to Research program (Dr action) but the research protocol number is not inserted (there is no Irpn action after Dr action). We denote this iteration by SkipResearchProt.

3. The third iteration in Dis; Tnn; Rtn, DNr; Ipd; Das, which is a legal iteration. We denote this iteration by NoteRun.

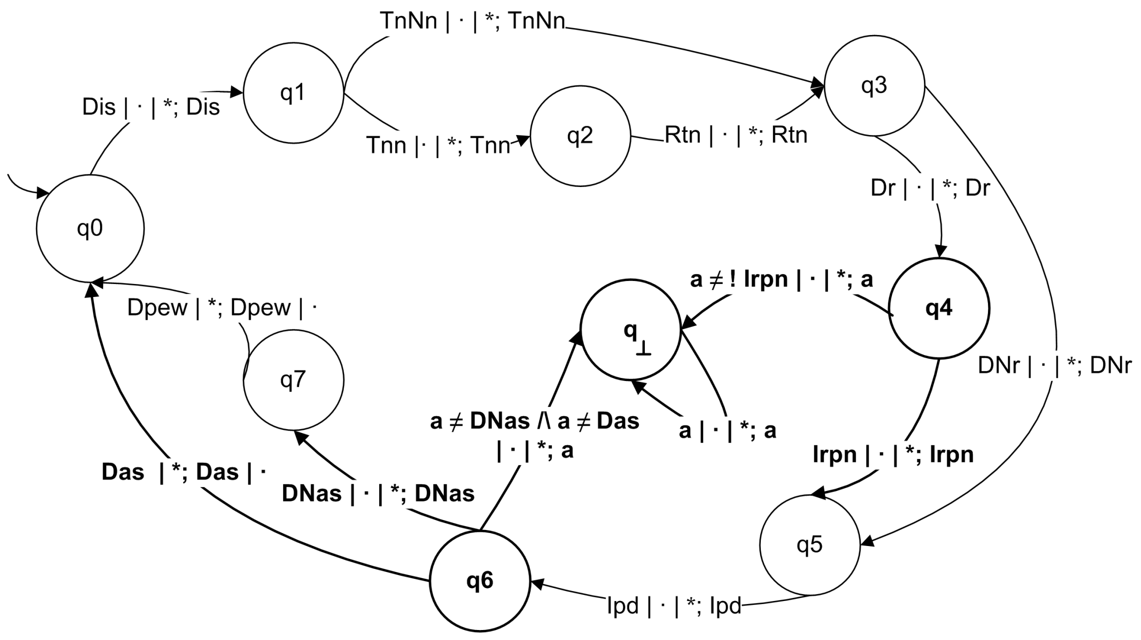


Figure 5.10: Longest-valid-prefix automaton constructed from the Policy automaton for drug selection process.

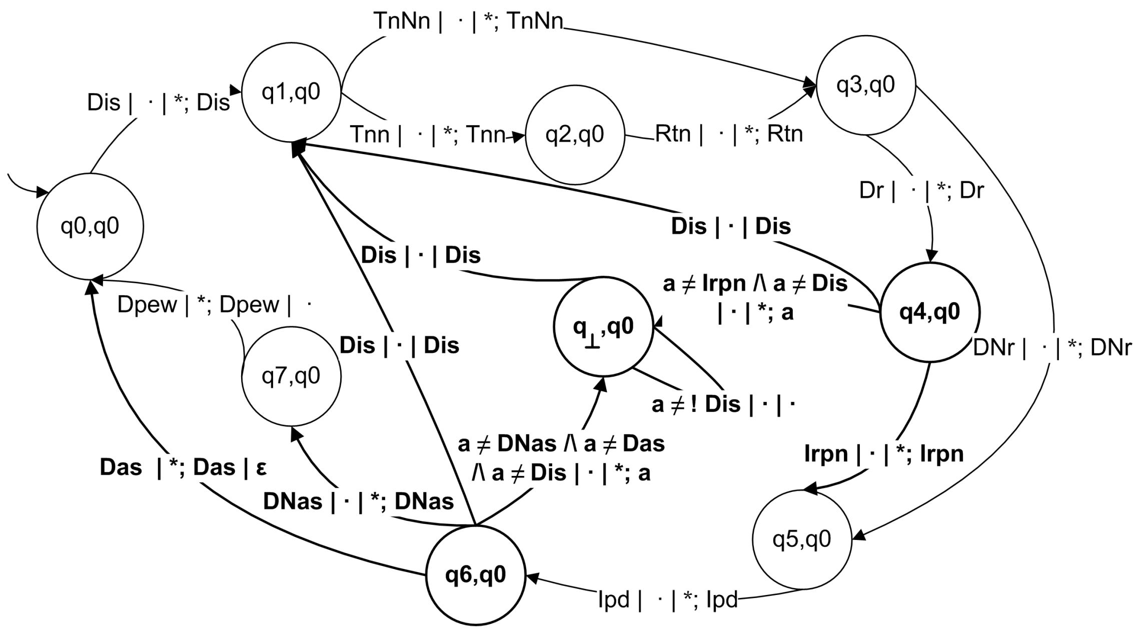


Figure 5.11: Iterative Suppression automaton constructed from the Policy automaton for drug selection process.

Table 5.2: Difference in enforcement of the drug selection process by Longest-valid-prefix automaton and Iterative Suppression automaton

No.	Input	Output	
		Longest-valid-prefix automaton from Figure 5.10	Iterative Suppression automaton from Figure 5.11
1	ResearchRun	ResearchRun	ResearchRun
2	SkipResearchProt	·	·
3	NoteRun	NoteRun	NoteRun
1; 2; 3	ResearchRun; SkipResearchProt; NoteRun	ResearchRun	ResearchRun; NoteRun

The resulting trace is illegal since it has an irremediably bad second part. How can the mechanism modify such execution? In Table 5.2 we show the results of transformation of these iterations and their concatenation.

5.4.5 Enforcement of an Anonymization Policy

We here show the difference between the constructions of enforcement mechanisms used in this chapter by means of enforcing some practical security policies. We discuss two examples of rather intuitive anonymization policy.

Example 5.4.1 *No non-anonymized personally identifiable data (pid) may leave the system without notification.*

Example 5.4.2 *No non-anonymized personally identifiable data (pid) may leave the system.*

Even though these to policies look similar, they have a completely different semantics. The first policy says that *some* non-anonymized pid is still allowed to leave the system, but the notification must be shown to the user. The second policy forbids any sending of non-anonymized pid. In our case study our colleagues from HSR would have a strong preference over these two policies: they rather prefer to have a policy that does not allow any leakage of non-anonymized pid (Example 5.4.2). We refer an interested reader to [37], where File F framework requires that the patients' data must be anonymized if previously requested by the patient. Then, the report containing anonymized data can be sent outside the hospital.

Let us first simplify the model of the system and its events just to give a brief example. Assume that the only events that are relevant to the security policies are:

SendA	Send anonymized data
SendN	Send non-anonymized data
Notif	Execution of notification

Enforcement of Example 5.4.1 policy

This policy can be enforced in different ways. One way is suppressing of events corresponding to sending non-anonymization data (SendN); another way is a modification of these events (for example, by anonymizing the data); and another way is an insertion of an event corresponding to the execution of a notification (Notif) as soon as the non-anonymized data is sent. All alternatives are possible, however not all of them are equally acceptable by the stake holders.

The security policy of Example 5.4.1 specifies that only sequences of a particular form $((\mathsf{SendN}; \mathsf{Notif})|\mathsf{SendA})^*$ are valid, hence sequences SendN; Notif and SendA are valid iterations. Assume that once a non-anonymized left the system without the notification (that violates the policy) and then the next data left the system anonymized. The sequence of actions corresponding to this example is SendN; SendA.

The Longest-valid-prefix automaton will output the longest valid prefix, which in this case is an empty trace. The Iterative Suppression automaton will recognize that the first event SendN is an unfinished trace, while the second event SendA is a trace itself, hence it will output SendA.

We can see from this example that Iterative Suppression automaton provides better enforcement of the given policy than the Longest-valid-prefix automaton, still we do not claim that this enforcement result is the optimal one. Another way to enforce the given policy is substitute sending of non-anonymized data by first anonymizing pid, and then sending it. In order to provide such an enforcement, our mechanism should be able to add events that were not in the original system beforehand, such as Anonym, which means "Anonymize the personally identifiable data". Then, an enforcement mechanism as a sequence transformer can potentially replace SendN with Anonym; SendA.

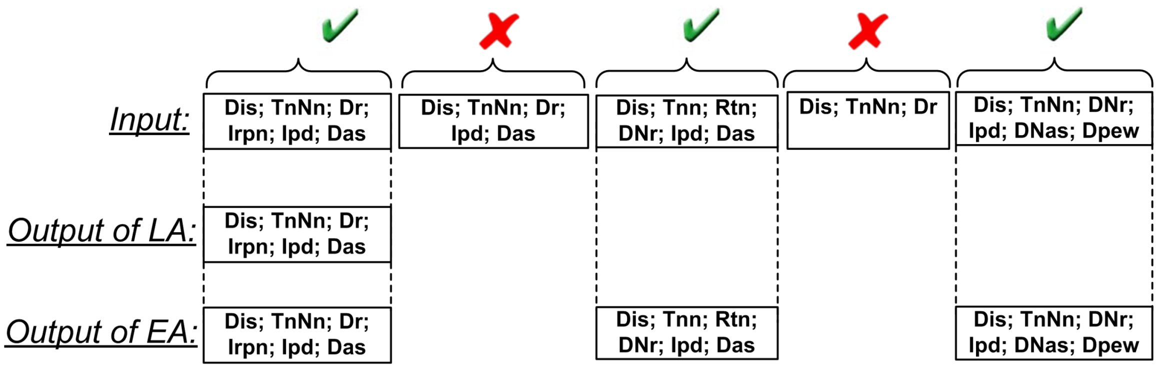

Input:
Output of LA:
Output of EA:
Dis; TnNn; Dr; Irpn; Ipd; Das
Dis; TnNn; Dr; Ipd; Das
Dis; Tnn; Rtn; DNr; Ipd; Das
Dis; TnNn; Dr
Dis; TnNn; DNr; Ipd; DNas; Dpew
Dis; TnNn; Dr; Irpn; Ipd; Das
Dis; TnNn; Dr; Irpn; Ipd; Das
Dis; Tnn; Rtn; DNr; Ipd; Das
Dis; TnNn; DNr; Ipd; DNas; Dpew

Figure 5.12: Output of the Longest-valid-prefix automaton (LA) and Iterative Suppression automaton (EA) on the same tentative execution sequence.

Enforcement of Example 5.4.2 policy

This policy can also be enforced in two ways: by suppression of SendN events, or by modification of these events (substitute SendN with Anonym; SendA). The security policy of Example 5.4.2 specifies that only sequences of the form SendA^* are valid. Like in previous example, the invalid sequence SendN; SendA will be transformed by a Longest-valid-prefix automaton into an empty sequence and by Iterative Suppression automaton into a sequence of one action SendA. Similarly to previous case, there is another way to enforce this policy by replacing SendN with Anonym; SendA. We shall come back to this example in the next chapter.

5.4.6 Discussion

In Figure 5.11 we partially show the result of the Iterative Suppression automaton construction for the same Policy automaton according to the semantics from Figure 5.8. For an easier comparison, we also emphasize the outcoming transitions from the states q_4, q_6 and $q_\perp$. This automaton also leads to an error state as soon as something bad happens, however it is able to recognize the beginning of a new good iteration which gives this automaton the power of producing more output for the same bad input.

Let us show in Figure 5.12 the output of the Longest-valid-prefix automaton and iterative suppression automaton for the same policy that is represented by a Policy automaton in Figure 5.3. The input contains 5 iterations corresponding to the drug selection process. The reader is already acquainted with the first 3 of them - they are the same as in the Section 5.4.4. Iterations 1, 3 and 5 are valid, and iterations 2 and 4 are invalid, hence the whole input is not valid and is irremediable. However, for iterations 1, 3 and 5 the Doctor managed to proceed successfully so there are three legal substrings: iterations 1, 3 and 5.

The Longest-valid-prefix automaton shown in Figure 5.10 outputs only the first iteration. It means that Doctor will successfully complete selection process only for the first drug. The Iterative Suppression automaton shown in Figure 5.11 will output all three successful iterations.

Future direction

As a more general approach, we can consider the case of actions that cannot be corrected. We call these actions *observable* actions. For instance, in a business processes they correspond to outsourced services, where actions can only be logged. It is possible to have observable actions also within an organization; for example when a doctor is preparing a set of drugs for a specific patient, he takes a wrong drug from a stock, and it is not possible to delete this physical action. It could be modeled as a special kind of Iterative suppression automaton that cannot suppress observable actions.

We can also consider the case of multiple users and define behavior of enforcement mechanism in that case. Indeed, many doctors may try to dispense drugs at the same time, and construction of enforcement mechanism can be different. We also leave this problem for future work.

5.5 Summary

In this chapter we first introduced additional notations and assumptions about the representation of the security policy made in this thesis: the security policy is presented as a Policy automaton, which is an automaton that combines the acceptance conditions of finite-state automaton and Büchi automaton.

We proposed a construction of the Longest-valid-prefix automaton from the security policy represented as a Policy automaton. The Longest-valid-prefix automaton is a particular kind of edit automaton that always outputs the longest valid prefix of the tentative execution sequence.

We have presented a construction of a new kind of edit automaton, called Iterative suppression automaton, that is able to recognize valid iterations of the tentative executions and suppress the actions between the valid iterations.

By providing these two constructions, we have addressed Issue 2: we have a new generic construction for two kinds of edit automata: one outputs the longest valid prefix of the input and another one outputs more than the longest valid prefix in case the execution sequence is invalid.

Chapter 6

Predictable Enforcement and Error-Toleration

The current theory of runtime enforcement is based on two notions: soundness and transparency. Soundness defines that the output is always good and transparency defines that good input is not changed. In this chapter we propose a new notion of predictability that ensures that there are "no surprises on bad input". We address the problem policy of enforcement when the users make insignificant errors and would like to continue the process execution. We present a novel approach for constructing enforcement mechanism that tolerate user errors.

6.1 Predictability of Enforcement

As we have seen in the example of market policy from Section 3.1, on the example of drug dispensation process from Section 3.2, and later example of simple anonymization policy from Section 5.4.5, the same security property can be enforced in different ways. Currently, the only formal notions that describe the behavior of enforcement mechanism are soundness and transparency (that together form a notion of effective$_=$enforcement).

In our opinion a new notion should be proposed that describes the behavior of runtime enforcer when the execution sequence is invalid (in case it's valid, it should not be changed according to transparency). In this section we propose a new notion that addresses Issue 3.

Recall the example of drug selection process from Section 3.2. The set of expected execution sequences corresponding to this process that were presented on page 36 form a security policy P.

SimpleRun Dis; TnNn; DNr; Dpres,

NoteRun Dis; Tnn; Rtn; DNr; Dpres,

ResearchRun Dis; TnNn; Dr; Irpn; Dpres,

NoteResearchRun Dis; Tnn; Rtn; Dr; Irpn; Dpres,

and their closure under concatenation: for every $\sigma, \sigma' \in P : \sigma; \sigma' \in P$. Notice that an empty trace **NoRun** also satisfies the policy.

Let us now make some examples of invalid traces with respect to the policy P.

Example 6.1.1 *The doctors might forgot to click the "I have read the Therapeutical Note" button and rather close the window (*Ctw*). A similar event could happen for the step in which research protocol numbers are not inserted (*Cpw*), or he might skip all steps altogether. For example, these alternatives give us the following traces*

CloseProt Dis*;* TnNn*;* Dr*;* Cpw*;* Dpres

SkipAll Dis*;* Dr*;* Dpres

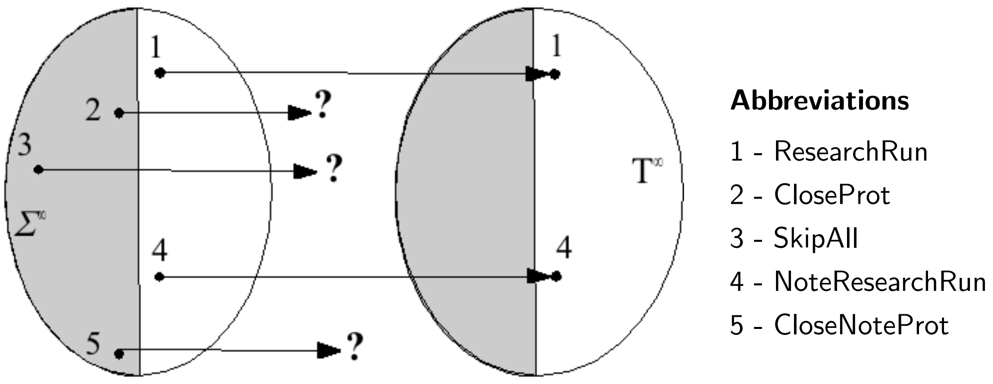
CloseNoteProt Dis*;* Tnn*;* Ctw*;* Dr*;* Cpw*;* Dpres

Figure 6.1 graphically shows a behavior of enforcement mechanism that is sound and transparent with respect to some security property $\widehat{P}$. Execution sequences that are intercepted by a runtime enforcer are shown on the left side of the figure and marked with Σ^∞. The resulting outputs of runtime enforcer are shown on the right side and marked with T^∞. The gray area denotes invalid sequences and the white area denotes valid ones with respect to $\widehat{P}$.

Transparency means that the traces in the white area of the input are mapped into *the same* traces (at the same position) in the white area of the output. Soundness means that all the traces shall be mapped into the white area. However, it is not specified where exactly the points from gray area are mapped.

The valid traces ResearchRun and NoteResearchRun are mapped into the same traces (numbers 1 and 4 in the figure), which corresponds to transparency. Invalid traces CloseProt, SkipAll and CloseNoteProt (number 2, 3 and 5 in the figure) are mapped into *some* good traces that can be chosen arbitrarily.

Notice that our definition of transparency uses a notion of identity relation instead of semantic-equvalence relation that was originally proposed [7] (see Section 4.3.3 for definitions of soundness and transparency). That is why valid traces are mapped into *the*



This mechanism is transparent since the valid executions ResearchRun and NoteResearchRun are not modified – they are in the same point in the output space T^∞. The mechanism is also sound, hence all the remaining invalid executions can be transformed into any point in the white (good) part of the output space.

Figure 6.1: Sound and Transparent enforcement mechanism.

same points in Figure 6.1. The reason for this decision is that it is not possible to define a semantics that can be used in all application domains. Recently, Khoury and Tawbi [47] discussed possible semantics of this relation, however they also proposed some concrete definitions for concrete domains. Ligatti and Reddy [52] have proposed the notion of completeness instead. It can be easily shown that transparency implies completeness and since transparency is necessary here, we don't discuss it further.

Figure 6.1 explicitly shows that the notions of soundness and transparency are not sufficient to distinguish between different enforcement mechanisms and to identify whether one mechanism provides a better enforcement than another one. We think that a new notion will help to answer these questions.

6.1.1 Metrics and Distances

In order to introduce a new notion we recall the definitions of metrics and metric spaces. These concepts in this thesis are adapted from [4, 29].

Definition 6.1.1 *A* metric *on a set S is a function $d : S \times S \to \mathbb{R} \cup \{\infty\}$ such that*

(a) $non-negativity : d(\sigma, \tau) \geq 0$,

(b) $identity : d(\sigma, \tau) = 0$ if and only if $\sigma = \tau$,

(c) symmetry: $d(\sigma, \tau) = d(\tau, \sigma)$,

(d) triangular inequality: $d(\sigma, \tau) \leq d(\sigma, \sigma') + d(\sigma', \tau)$.

The pair (S, d) is called a *metric space* and the number $d(\sigma, \tau)$ is called the *distance between the elements* σ and τ. If all conditions but symmetry hold, then d is called a *quasi-metric.*

When a distance is a finite number we can compare how far two situations (some potentially illegal traces) are from the legal situation. For medical staff these two situations can be perceived as qualitatively similar with a different degree of gravity. The notion of ∞ can be used to represent distance between traces perceived so qualitatively different that they are incomparable. An example: a trace containing a wrong action compromising the health of a patient is incomparable to any legal trace.

We discuss here some concrete metrics and quasi-metrics that will be useful in comparing tentative execution sequences with the expected behavior of our running examples.

Definition 6.1.2 *A* suppressing distance $d_S(\sigma, \sigma')$ *between two finite traces* σ *and* σ' *is a number of actions that must be suppressed from* σ *in order to get* σ'*. The total function* $d_S : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ *is a quasi-metric, such that*

$$d_S(\sigma, \sigma') = \begin{cases} \infty & \text{if } \sigma = \cdot \text{ and } \sigma' \neq \cdot \\ |\sigma| & \text{if } \sigma' = \cdot \\ d_S(\sigma_a, \sigma'_a) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + d_S(\sigma_a, b\sigma_b) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = b\sigma_b \text{ and } a \neq b \end{cases} \tag{6.1}$$

Notice that the distance between sequences σ and σ' is ∞ when $|\sigma| < |\sigma'|$ because no number of suppressions can transform σ into σ'. Another non-trivial rule is the last case in the equation 6.1: if the first actions of the two sequences are different then the action in the first sequence is suppressed and comparison continues.

The intuition behind suppression is that a bad trace is close to a good trace if the actions we have to undo are few.

Let us come back to our case study of the drug selection process (more generically, we can also consider other workflows of the hospital). The idea of suppression distance can be explained to the operator in the hospital and can be acceptable for an administrative procedure (albeit annoying for the operator involved). For example, the monitor could block the process if the number of bad actions exceeds a given threshold or, preferably, it could undo all bad actions bringing us back to the point where we started the transaction that has gone awry. More concretely, see the following example.

Example 6.1.2 *A doctor is selecting a drug (*Dis*) for which therapeutical notes are needed (*Tnn*). However at the time to review them, he is interrupted (e.g. in case of an emergency, interruption by another colleague etc.). When he comes back, he has to start running the process again because it timed out. The second time the doctor successfully finishes the process (he executes the sequence* NoteResearchRun*). So, his tentative execution σ is* Dis*;* Tnn*;* NoteResearchRun *and the distance to one of the good traces $\sigma'=$* NoteRun *is*

$$d_S((\mathsf{Dis};\mathsf{Tnn};\mathsf{NoteResearchRun}),\mathsf{NoteResearchRun})=2 \tag{6.2}$$

Notice that the distance from the legal sequence NoteResearchRun *to the illegal tentative sequence* Dis*;* Tnn*;* NoteResearchRun *is ∞ because no number of suppressions can transform the legal trace into the illegal one.*

Similarly, the distance between this tentative sequence and another legal sequence, for example the distance between σ = Dis; Tnn; NoteResearchRun *and σ' =* NoteRun= Dis; Tnn; Rtn; DNr; Dpres*, is also ∞ because tentative execution is shorter than the expected legal one.*

In the next example we can see that even suppressing distance already discriminates between different run-time enforcement mechanisms.

Example 6.1.3 *If an enforcement mechanism used to enforce the drug selection process is the Longest-valid prefix automaton (marked as E_{LA}), then it will transform every tentative execution sequence to its longest valid prefix, hence for tentative sequence* Dis; Tnn; NoteRun *(from Example 6.1.2), we have E_{LA}(*Dis; Tnn; NoteRun*) = $\cdot$. The expected behavior in this case is a sequence* NoteRun*, and E_{LA}(* NoteRun*) =* NoteRun*. Hence, if we compare the results of the Longest-valid-prefix automaton E_{LA} for tentative execution sequence and for an expected one, we have:*

$$d_S(E_{LA}(\mathsf{Dis};\mathsf{Tnn};\mathit{NoteRun}),E_{LA}(\mathsf{NoteRun}))=d_S(\cdot,\mathsf{NoteRun})=\infty \tag{6.3}$$

An Iterative Suppression automaton (marked as E_{IS}, see Section 5.4 for more details) would have suppressed the illegal initial prefix and recognize a valid iteration NoteRun*: $E_{IS}(\sigma)=E_{IS}$(*Dis; Tnn; NoteRun*) =* NoteRun*, and the suppressing distance then is:*

$$d_S(E_{IS}(\mathsf{Dis};\mathsf{Tnn};\ \mathsf{NoteRun}),E_{IS}(\ \mathsf{NoteRun}))=d_S(\mathsf{NoteRun},\ \mathsf{NoteRun})=0 \tag{6.4}$$

To distinguish between more enforcement mechanisms we use another metric that counts the number of replaced actions.

Let us come back again to the example of drug dispensation process. While enforcing a process in the hospitals, an enforcement mechanism could be entitled only to correcting small errors without changing the protocol used by the operators (such as patient identification, patient consent and blood sampling before blood transfusion). If the doctor forgot to fill one field in the form, the mechanism can insert a default value. On the other hand, insertions of new steps by the enforcement mechanism to compensate a bad event are not be allowed because a different protocol might have different medical or legal consequences and those can only be judged by an expert.

Definition 6.1.3 *The* replacing distance $d_R(\sigma, \sigma')$ *between two finite traces σ and σ' is a number of replacements that should be done in σ in order to get σ'. The total function $d_R : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ is a metric, such that*

$$d_R(\sigma, \sigma') = \begin{cases} 0 & \text{if } \sigma = \cdot \text{ and } \sigma' = \cdot \\ \infty & \text{if } \sigma = \cdot \text{ xor } \sigma' = \cdot \\ d_R(\sigma_a, \sigma'_a) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + d_R(\sigma_a, \sigma_b) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = b\sigma_b \text{ and } a \neq b \end{cases} \tag{6.5}$$

Notice that the distance is ∞ if the tentative execution sequence and the expected one have different length (it is expected because they clearly belong to different protocols).

A more general definition of distance was originally proposed by Levenshtein [50].

Definition 6.1.4 *The* Levenshtein distance $d_L(\sigma, \sigma')$ *between two finite traces σ and σ' is a number of insertions, suppressions and replacements needed to obtain σ' from σ. The total function $d_L : \Sigma^* \times \Sigma^* \to \mathbb{N}$ is a metric, such that*

$$d_L(\sigma, \sigma') = \begin{cases} |\sigma'| & \text{if} \quad \sigma = \cdot \\ |\sigma| & \text{if} \quad \sigma' = \cdot \\ d_L(\sigma_a, \sigma'_a) & \text{if} \quad \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + \min\{d_L(\sigma_a, \sigma_b), & \text{if} \quad \sigma = a\sigma_a \text{ and} \\ d_L(a\sigma_a, \sigma_b), d_L(\sigma_a, b\sigma_b)\} & \quad\;\; \sigma' = b\sigma_b \text{ and } a \neq b \end{cases} \tag{6.6}$$

Let us explain the difference between the Levenshtein distance and the replacing distance by means of example.

Example 6.1.4 *Consider a case from Example 6.1.1, where the doctor forgot to read the therapeutical notes and closed the window (he performed* Ctw *action instead of* Rtn*) and the doctor did not insert the research protocol number (he performed* Cpw *instead of* Irpn*), the corresponding tentative execution sequence is*

CloseNoteProt = Dis; Tnn; Ctw; Dr; Cpw; Dpres.
Suppose that an enforcement mechanism E leaves this sequence without changes. We assume that E is transparent, so an already legal execution sequence
NoteResearchRun = Dis; Tnn; Rtn; Dr; Irpn; Dpres.
is not changed by E. Hence, if we compare the output of an enforcement mechanism E by means of replacing distance d_R and Levenshtein distance d_L, we get:

$$d_R(E(\mathsf{CloseNoteProt}), E(\mathsf{NoteResearchRun})) = d_R(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$$

$$d_L(E(\mathsf{CloseNoteProt}), E(\mathsf{NoteResearchRun})) = d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$$

Replacing distance in this case is equal to the Levenshtein distance.

Example 6.1.5 *Let us consider the sequence* CloseProt = Dis; TnNn; Dr; Cpw; Dpres*(where the doctor only did not insert a research protocol number). Assume that an enforcement mechanism E does not change this execution sequence, and it is transparent, so it does not change the legal execution* NoteResearchRun = Dis; Tnn; Rtn; Dr; Irpn; Dpres.
Then,

$$d_R(E(\mathsf{CloseProt}), E(\mathsf{NoteResearchRun})) = d_R(\mathsf{CloseProt}, \mathsf{NoteResearchRun}) = \infty$$

because NoteResearchRun is simply longer than CloseProt (and belongs to another process execution where theraperutical notes are not needed). But the Levenshtein distance counts the inserted and replaced actions and the distance is

$$d_L(E(\mathsf{CloseProt}), E(\mathsf{NoteResearchRun})) = d_L(\mathsf{CloseProt}, \mathsf{NoteResearchRun}) = 3$$

We present the Levenshtein distances between the execution sequences for our running example of drug selection process in Table 6.1.

6.1.2 From Sound to Bounded Enforcement Mechanisms

Building on metrics over the sequences, we propose to use the notions from measure theory for enforcement mechanisms. In the following definitions $(\Sigma^\infty \cup \mathrm{T}^\infty, d)$ is a metric space, a map $E : \Sigma^\infty \to \mathrm{T}^\infty$ is an enforcement mechanism and a security policy is a set $P \subseteq \Sigma^\infty \cap \mathrm{T}^\infty$. As a starting point we assume that all runtime enforcers satisfy the transparency property.

In the context of our running example, transparency of an enforcement mechanism has the following meaning. The actions in the systems corresponds to actions of doctors and

Table 6.1: The Levenshtein distances between some sequences

	NoRun	ResearchRun	CloseProt	SkipAll	NoteResearchRun	CloseNoteProt
NoRun	0	5	5	3	6	6
ResearchRun	5	0	1	2	2	3
CloseProt	5	1	0	2	3	2
SkipAll	3	2	2	0	3	3
NoteResearchRun	6	2	3	3	0	2
CloseNoteProt	6	3	2	3	2	0
NoteRun	5	3	3	3	2	3

nurses (or administrative staff), who are knowledgeable and accountable for their actions. Their point of view is that the choice of a *legitimate* course of action is due to a contextual knowledge not available to the system. A system that would change their decisions, when those actions conform to the policy of the hospital, would be unacceptable.

The next step is generalizing the notion of soundness. We start from a classical definition of bounded map that Figure 6.2 shows graphically. Even though the division of traces into good and bad is not relevant for the definition of boundedness, we keep it in the figure to ease the comparison with other notions.

Definition 6.1.5 *A function $E : \Sigma^\infty \to \mathrm{T}^\infty$ is* bounded *if the subset $\{E(\sigma) : \sigma \in \Sigma^\infty\} \subseteq \mathrm{T}^\infty$ is bounded. Formally,*

$$\exists \tau \in \mathrm{T}^\infty : \exists \varepsilon > 0 : \forall \sigma \in \Sigma^\infty : d(E(\sigma), \tau) \leq \varepsilon \tag{6.7}$$

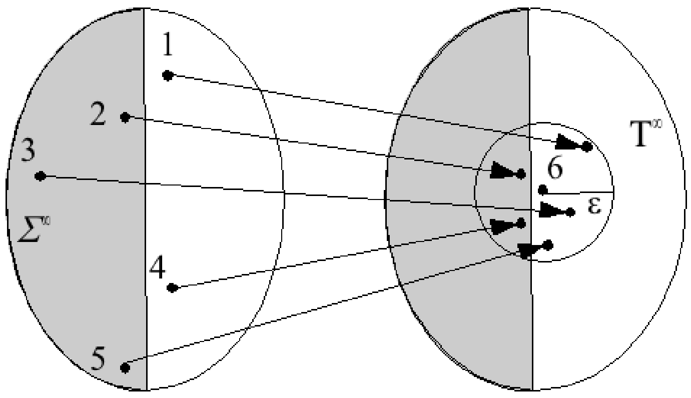
Let us project this notion to the theory of enforcement mechanisms. We get a mechanism that transforms all sequences to some sequence close to τ. This is not what users are expecting. The user's policy usually contains several good sequences, hence we should adapt the definition to map bad sequences to different good sequences in the policy.

Definition 6.1.6 *An enforcement mechanism $E : \Sigma^\infty \to \mathrm{T}^\infty$ is* bounded within ε *if and only if*

$$\exists \sigma_P \in P : \forall \sigma \in \Sigma^\infty : d(E(\sigma), E(\sigma_P)) \leq \varepsilon. \tag{6.8}$$

This notion says that an output of enforcement mechanism is always within the distance ε from some good execution, for $\varepsilon > 0$ we are weakening the notion of soundness. Figure 6.3 shows the bounded within ε enforcement mechanism.

There is a fundamental difference between boundedness and boundedness within ε. Boundness means that there is one single trace τ in the possible outputs such that *all*

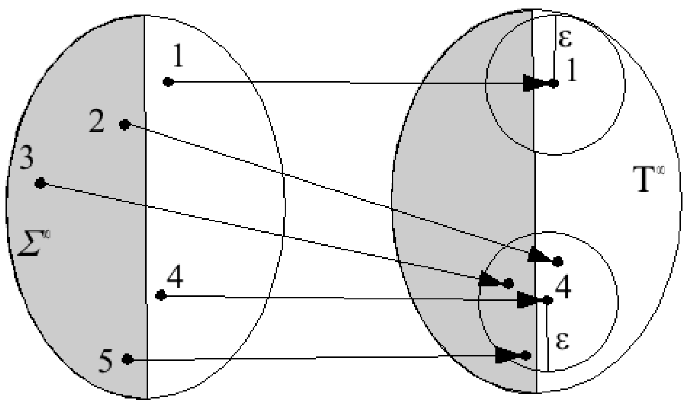


Abbreviations

1 - ResearchRun
2 - CloseProt
3 - SkipAll
4 - NoteResearchRun
5 - CloseNoteProt
6 - NoteRun

This enforcement mechanism is bounded because there exists a point 6 in the output space such that all the outputs of the enforcement mechanism are within some radius from this point.

Figure 6.2: Bounded map.



Abbreviations

1 - ResearchRun
2 - CloseProt
3 - SkipAll
4 - NoteResearchRun
5 - CloseNoteProt

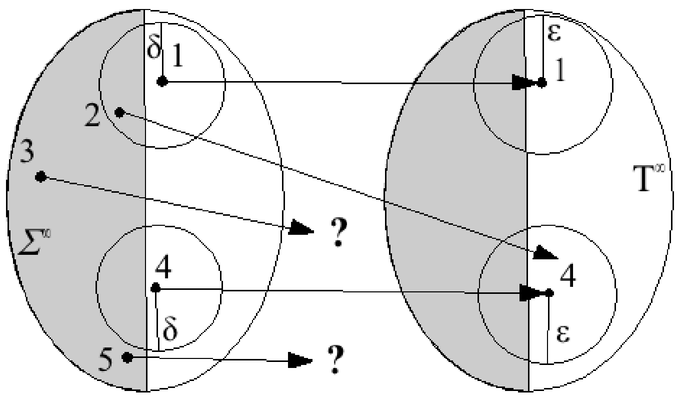
This mechanism is bounded within ε since for every execution of the process there exists some point in the output space that is at distance ε from some valid point (in this case it's either 1 or 4).

Figure 6.3: Bounded within ε enforcement mechanism.

the outputs of E are in the radius ε from τ. Boundedness within ε means that for every possible output there is a valid trace such that the distance between them is smaller or equal than ε.

Example 6.1.6 *An enforcement mechanism E enforces the drug dispensation process in the following way :*

$$E(\mathsf{CloseNoteProt}) = \mathsf{CloseNoteProt}$$
$$E(\mathsf{SkipAll}) = \mathsf{CloseNoteProt}$$
$$E(\mathsf{CloseProt}) = \mathsf{NoteRun}$$



Abbreviations

1 - ResearchRun
2 - CloseProt
3 - SkipAll
4 - NoteResearchRun
5 - CloseNoteProt

This mechanism is conditionally bounded within ε because only point 2 that is close to the valid point 1, is mapped into a point within ε from some valid point 4. Notice that this mapping is allowed by the definition. The other points, that are not close enough to 1 or 4, can be mapped to arbitrary valid points (just to satisfy soundness).

Figure 6.4: Conditionally bounded within ε enforcement mechanism.

The smallest distance from $E(\sigma)$ to some good sequence is:

$$d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$$
$$d_L(\mathsf{NoteRun}, \mathsf{NoteResearchRun}) = 2$$

Since the output of E is always at distance 2 from some good execution ***NoteResearchRun*** *then E is bounded within $\varepsilon = 2$.*

We can refine the notion by imposing also that "almost valid traces" should be mapped to some "almost valid traces". We call it *conditional boundedness within ε* and show graphically in Figure 6.4.

Definition 6.1.7 *An enforcement mechanism $E : \Sigma^\infty \to \mathrm{T}^\infty$ is* conditionally bounded within ε *if and only if*

$$\exists\delta > 0 : \forall\sigma \in \Sigma^\infty : (\exists\sigma'_P \in P : d(\sigma, \sigma'_P) \leq \delta \Rightarrow \exists\sigma_P \in P : d(E(\sigma), E(\sigma_P)) \leq \varepsilon). \quad (6.9)$$

Example 6.1.7 *An enforcement mechanism E transforms the sequences of actions in the following way :*

$$E(\mathsf{CloseNoteProt}) = \mathsf{CloseProt}$$
$$E(\mathsf{SkipAll}) = \mathsf{NoRun}$$
$$E(\mathsf{CloseProt}) = \mathsf{NoteRun}$$

The smallest distance from $E(\sigma)$ to some good sequence is:

$$d_L(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1$$
$$d_L(\mathsf{NoRun}, \mathsf{NoRun}) = 0$$
$$d_L(\mathsf{NoteRun}, \mathsf{NoteRun}) = 0$$

Obviously, E is bounded within $\varepsilon = 1$. It is conditionally bounded within $\varepsilon = 1$ because there is a $\delta = 3$ such that for every sequence there is a valid trace at most at distance 3:

$$d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$$
$$d_L(\mathsf{SkipAll}, \mathsf{NoRun}) = 3$$
$$d_L(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1$$

Boundedness and conditional boundedness can refine soundness but are still unacceptable: when a doctor performs an execution CloseProt where he closes the window instead of inserting the protocol (instead of executing ResearchRun), an enforcement mechanism may transform his tentative execution CloseProt into another, completely different one (execution NoteRun). The problem is that some actions in the outcome are not a direct transformation of the actions of the doctors. Since actions carry liabilities it is important that the modifications made by an enforcement mechanism are always linked to corresponding actions by the doctor.

6.1.3 Predictability

Our notion of predictability within ε is inspired by the classical notion of continuous functions. Let $(\Sigma^\infty \cup \mathrm{T}^\infty, d)$ and $(\Sigma^\infty \cup \mathrm{T}^\infty, d')$ be metric spaces.

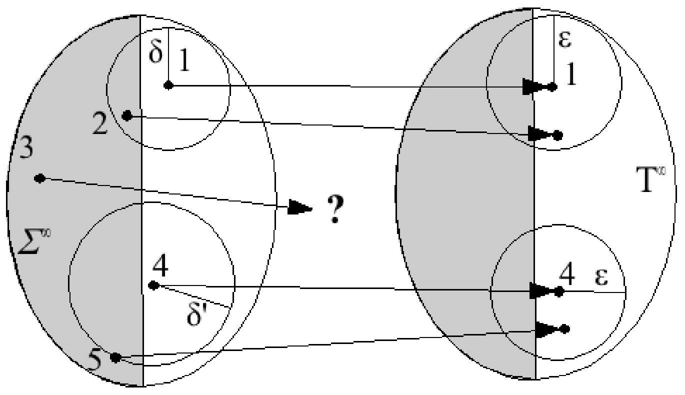
Definition 6.1.8 *A map $E : \Sigma^\infty \to \mathrm{T}^\infty$ is* continuous *if at every trace $\sigma \in \Sigma^\infty$ the following holds:*

$$\forall \varepsilon > 0 : \exists \delta > 0 : \forall \sigma' \in \Sigma^\infty : (d(\sigma, \sigma') < \delta \Rightarrow d'(E(\sigma), E(\sigma')) < \varepsilon) \tag{6.10}$$

In conditional boundedness we proposed to limit an output when an input is "almost good". But as we have seen from Figure 6.4, an input and its output should be compared to *the same* valid trace.

Definition 6.1.9 *An enforcement mechanism E is* predictable within ε *if for every trace $\sigma_P \in P$ the following holds:*

$$\forall \nu \geq \varepsilon : \exists \delta > 0 : \forall \sigma \in \Sigma^* : (d(\sigma, \sigma_P) \leq \delta \Rightarrow d'(E(\sigma), E(\sigma_P)) \leq \nu) \tag{6.11}$$



Abbreviations

1 - ResearchRun
2 - CloseProt
3 - SkipAll
4 - NoteResearchRun
5 - CloseNoteProt

We graphically show an execution sequence transformer that is predictable within ε. Indeed, we have two valid executions: 1 and 4. Whenever a tentative execution is within some range δ from a valid execution, e.g. 2 is in the range of 1, it is transformed into an execution within ε from this valid execution, e.g. 2 is transformed into a sequence close to 1. If an execution is "too far" from being valid, e.g. execution 3, predictability does not define the way in which it should be transformed.

Figure 6.5: Predictable within ε enforcement mechanism

Informally, it says that for every valid trace there always exists a radius δ, such that all the traces within this radius are mapped into the circle with radius ε from this trace. We show it in Figure 6.5.

6.1.4 Examples of Enforcement Mechanisms

Enforcement mechanisms for market policy

Let us come back to the example of market policy from Example 3.1.1 and two enforcement mechanisms for it:

- Longest-valid-prefix automaton from Section 5.3 (we denote it by E_{LA});
- Iterative Suppression automaton from Section 5.4 (we denote it by E_{IS}).

First we analyze whether the Longest-valid-prefix automaton E_{LA} is predictable within some bound k and then try to find k. Without loss of generality, we will use the Levenshtein distance in these examples.

Recall that valid iterations for this policy are of the form (`take(n)`;`pay(n)`) , or `browse`, or (`pay(n)`;`browse`*;`take(n)`).

Consider a tentative execution $\sigma =$ `take(1)`;(`pay(2)`;`take(2)`)^n. The Longest-valid-prefix automaton always outputs the longest valid prefix of its input, hence $E_{LA}(\sigma) = \cdot$.

By the shape of σ we can see that there exists a legal trace $\sigma_P = (\texttt{pay(2)};\texttt{take(2)})^n$ such that

$$d_L(\sigma, \sigma_P) = d_L(\texttt{take(1)};(\texttt{pay(2)};\texttt{take(2)})^n, (\texttt{pay(2)};\texttt{take(2)})^n) = 1 \leq \delta$$

and this statement holds for all $\delta > 0$.

Then, according to the definition of predictability, if there exists a bound ε, then for all $\nu \geq \varepsilon$ the following should hold: $d_L(E_{LA}(\sigma), E_{LA}(\sigma_P)) \leq \nu$. However,

$$\begin{aligned}
&d_L(E_{LA}(\sigma), E_{LA}(\sigma_P)) = \\
&d_L(E_{LA}(\texttt{take(1)};(\texttt{pay(2)};\texttt{take(2)})^n), E_{LA}((\texttt{pay(2)};\texttt{take(2)})^n)) = \\
&d_L(\cdot, (\texttt{pay(2)};\texttt{take(2)})^n) = 2n
\end{aligned}$$

Since n is not bounded, there is no bound ε for this mechanism, so this mechanism is not predictable. It also matches the intuition behind predictability: an input σ is close (within δ) from a valid trace σ_P, but $E_{LA}(\sigma)$ is not close to the same valid trace.

Let us now consider an Iterative Suppression automaton E_{IS}. The same tentative execution $\sigma = \texttt{take(1)};(\texttt{pay(2)};\texttt{take(2)})^n$, is transformed into its longest subpart that is correct: $E_{IS}(\sigma) = (\texttt{pay(2)};\texttt{take(2)})^n$. Again, this execution is at distance 1 from a valid trace $\sigma_P = (\texttt{pay(2)};\texttt{take(2)})^n$, and $d_L(\sigma, \sigma_P) = 1 \leq \delta$ for any $\delta > 0$.

We can find a bound ε for this enforcement mechanism if we make one assumption: we assume that the distance we are using counts only a number of different elements that are outside of the valid iterations. Let us establish a bound $\varepsilon = k$, then all the traces that have up to k invalid actions (k actions outside of the valid iterations), like $\sigma = \texttt{take(1)}_1;\ldots;\texttt{take(1)}_k;(\texttt{pay(2)};\texttt{take(2)})^n$, are transformed into $\sigma_P = (\texttt{pay(2)};\texttt{take(2)})^n$, therefore

$$\begin{aligned}
&d(E_{IS}(\sigma), E_{LA}(\sigma_P)) = \\
&d(E_{IS}(\texttt{take(1)}_1;\ldots;\texttt{take(1)}_k;(\texttt{pay(2)};\texttt{take(2)})^n), E_{IS}((\texttt{pay(2)};\texttt{take(2)})^n)) = \\
&d((\texttt{pay(2)};\texttt{take(2)})^n, (\texttt{pay(2)};\texttt{take(2)})^n) = 0
\end{aligned}$$

and hence, there always exists $\delta = \varepsilon = k$ such that equation (6.11) holds.

Enforcement mechanism for drug selection process

Recall the enforcement mechanism E from Example 6.1.7:

$$\begin{aligned}
&E(\mathsf{CloseNoteProt}) = \mathsf{CloseProt}\\
&E(\mathsf{SkipAll}) = \mathsf{NoRun}\\
&E(\mathsf{CloseProt}) = \mathsf{NoteRun}
\end{aligned}$$

It was shown that this mechanism is conditionally bounded within $\varepsilon = 1$. However, it is not predictable within $\varepsilon = 1$ because there exists $\sigma_P = \mathsf{ResearchRun}$ such that

$$\begin{aligned}
\exists \nu = 2 : \forall \delta > 0 : \ & \exists \sigma = \mathsf{CloseProt} :\\
& (d_L(\sigma, \sigma_P) = d_L(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1 \leq \delta \wedge\\
& d_L(E(\sigma), E(\sigma_P)) = d_L(E(\mathsf{CloseProt}), E(\mathsf{ResearchRun})) =\\
& d_L(\mathsf{NoteRun}, \mathsf{ResearchRun}) = 3 > 2 = \nu)
\end{aligned}$$

which means that

$$\exists \nu = 2 : \forall \delta > 0 : \exists \sigma \in \Sigma^* : (d(\sigma, \sigma_P) \leq \delta \wedge d(E(\sigma), E(\sigma_P)) > \nu)$$

Enforcement mechanism for an anonymization policy

Recall an anonymization policy from Example 5.4.2: "No non-anonymized personally identiable data (pid) may leave the system." Formally, we describe this policy as a set of traces of the form SendA^*. We have proposed several enforcement mechanisms for this policy:

1) whenever a non-anonymized pid is about to be sent, an enforcement mechanism suppressed this event: the SendN event is suppressed. This can be done either by the Longest-valid-prefix automaton (when the subsequent events are suppressed as well), or by an Iterative Suppression automaton (subsequent events are not suppressed is they are SendA).

2) whenever a a non-anonymized pid is about to be sent, an enforcement mechanism suppresses this event, anonymizes the pid and then send it (SendN is substituted with Anonym; SendA).

We denote the Iterative Suppression automaton with E_{IS}. Consider a tentative execution sequence $\sigma = \mathsf{SendN}; \mathsf{SendA}; \mathsf{SendA}; \mathsf{SendN}; ...; \mathsf{SendA}$ with k actions SendN and m

actions SendA. This sequence represents a general case for any illegal sequence since only two actions SendA and SendN are relevant. For Iterative Suppression automaton we will use the suppressing distance d_S.

An Iterative Suppression automaton will suppress all bad iterations, i.e. all actions SendN, hence

$$E_{IS}(\sigma) = E_{IS}(\mathsf{SendN}; \mathsf{SendA}; \mathsf{SendA}; \mathsf{SendN}; ...; \mathsf{SendA}) = \mathsf{SendA}; \mathsf{SendA}; ...; \mathsf{SendA} \quad (6.12)$$

where the output sequence contains m actions SendA. Let us denote the output sequence with σ_P. Then,

$$\begin{aligned}
&d_S(E_{IS}(\sigma), E(\sigma_P)) = \\
&d_S(E_{IS}(\mathsf{SendN}; \mathsf{SendA}; \mathsf{SendA}; \mathsf{SendN}; ...; \mathsf{SendA}), E_{IS}(\mathsf{SendA}; \mathsf{SendA}; ...; \mathsf{SendA})) = \\
&d_S(E_{IS}(\mathsf{SendA}; \mathsf{SendA}; ...; \mathsf{SendA}), E_{IS}(\mathsf{SendA}; \mathsf{SendA}; ...; \mathsf{SendA})) = 0
\end{aligned}$$

Therefore we conclude that

$$\forall \sigma \in \Sigma^* : (d_S(\sigma, \sigma_P) \leq \delta \Rightarrow d_S(E_{IS}(\sigma), E_{IS}(\sigma_P)) = 0 \leq \nu) \quad (6.13)$$

where ν is an arbitrary positive number, and we can pick any δ, this statement holds for all δ.

However, we get a different result if we use Levenshtein distance d_L. Since we suppress all the actions SendN, the distance between the outputs of $E_{IS}(\sigma)$ and $E_{IS}(\sigma')$ will not be 0, but rather will be equal to the distance between the original sequences σ and σ'.

Now let us consider an enforcement mechanism E described in case 2). Any tentative execution sequence of the form $\sigma = \mathsf{SendN}; \mathsf{SendA}; \mathsf{SendA}; \mathsf{SendN}; ...; \mathsf{SendA}$ with k actions SendN and m actions SendA. We shall consider Levenshtein distance d_L in this example. Tentative sequence σ is transformed as follows by E:

$$\begin{aligned}
&E(\sigma) = E(\mathsf{SendN}; \mathsf{SendA}; \mathsf{SendA}; \mathsf{SendN}; ...; \mathsf{SendA}) = \\
&(\mathsf{Anonym}; \mathsf{SendA}); \mathsf{SendA}; \mathsf{SendA}; (\mathsf{Anonym}; \mathsf{SendA})...; \mathsf{SendA}
\end{aligned}$$

where the output sequence contains k actions Anonym and $k + m$ actions SendA.

The biggest distance between the outputs of an enforcement mechanism in this example is $2k$. Assume that we compare the tentative execution σ with k actions SendN and m actions SendA to a legal execution σ_P with only m actions SendA. Then, $d_L(\sigma, \sigma_P) = k$. Our enforcement mechanism substitutes every SendN with Anonym; SendA, wich means

Table 6.2: Properties of enforcement mechanism.

Name	Pre-Condition	Post-Condition
Soundness		for every trace the output is always some valid trace
Transparency	if input is a valid trace	output is *the same* valid trace
Bounded Map		there is one valid trace such that output is always within ε from that trace
Boundedness		output is always within ε from some valid trace
Conditional Boundedness	if input is within δ from some valid trace	output is within ε from some valid trace
Predictability	if input is within δ from a valid trace	output is within ε from *the same* valid trace

that $E(\sigma)$ contains k actions Anonym and $k+m$ actions SendA, hence $d_L(E(\sigma), E(\sigma')) = 2k$.

Notice that $E(\sigma)$ always contains some actions Anonym and some actions SendA, hence this mechanism will always enlarge the tentative execution σ by k elements, whenever σ contains k actions SendN. Therefore, we can conclude that for a given ε we can always find $\delta = \varepsilon/2$, so the enforcement mechanism discussed above is predictable.

6.1.5 Discussion

Table 6.2 shows all the notions so far and the new notion of predictability. The main difference is that predictability has a pre-condition and post-condition that compare invalid traces to *the same* valid trace.

Limitations

An apparent limitation of our work is that we don't deal with infinite traces. We have actually considered some mathematical functions over infinite traces from [27] but we found a practical obstacle: *the end of the fiscal year effectively terminates any trace in the eyes of our stakeholders.*

Let us discuss an example of a natural distance from [27] that has a clear mathematical intuition for our domain and allow to obtain finite numbers when comparing infinite traces. We can use an economic approach and consider the discounted distance that discounts the remote differences in the sequence :

Definition 6.1.10 *The* discounted distance *between two infinite traces is a number calculated using a total function $d_D : \Sigma^\omega \times \Sigma^\omega \to \mathbb{N}$, such that*

$$d_D(\sigma, \sigma') = \begin{cases} |\sigma| & \text{if } \sigma' = \cdot \\ |\sigma'| & \text{if } \sigma = \cdot \\ d_D(\sigma_a, \sigma'_a) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + \frac{1}{k}\, d_D(\sigma_a, \sigma_b) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = b\sigma_b \text{ and } a \neq b \end{cases} \tag{6.14}$$

In this definition $k \in \mathbb{R}$, $k \neq 0$ is a discounting factor.

The first function can be acceptable from the perspective of a risk manager as later events have less risk of being detected or of having consequences within the year. It is less acceptable for doctors: a wrong drug is a wrong drug, no matter if delivered at the beginning or the end of the fiscal year.

At the end the only acceptable metrics boils down to considering what happens during a limited slot and project it to infinity. But then we could simply consider the finite slot. Hence, the definition of mathematically simple and meaningful metrics for infinite traces is still open for us (assuming we should consider them at all).

Sequence alignment in bioinformatics

In bioinformatics the comparison of two sequences of elements is called *alignment.* Sequence alignment compares sequences like DNA or RNA and identifies regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [56].

There is a large variety of alignment methods in bioinformatics. Exact alignment methods done by dynamic programming are using constants of functions costs to the following events: insertion, suppression, replacements and also matching between the two elements of sequences. In these methods the distance between the sequences is a similarity score between them (the higher the score, the better is matching). Another group of alignment methods, called hybrid methods are used in cases when the downstream part of one sequence overlaps with the upstream part of the other sequence. This case is not useful in enforcement theory because we compare a valid sequence with invalid sequence that contains some errors.

Suppression, replacement and even Levensein distance between the sequences is not used in bioinformatics because these distances are purely syntactical (the cost of suppressing one element of the sequence is *exactly* the same as the cost of suppressing any other element). In practice, to compute the similarity between two sequences like DNA or RNA, either exact or hybrid methods are used with costs or functions that have some semantics (for example, according to what has already been observed and statistically measured between the organism(s) from which the sequences come from).

Other definitions of distances

So far we have used syntactical distances in our work. This approach is useful in case when suppressing or replacing one action in the execution sequence has the same cost as suppressing or replacing any other action. Our work can be extended using the constants of functions costs assigned to actions. For example, in case of execution traces of business processes, the risk manager can assign the costs to the action of her business process that represent the risk associated with this action. In general, we think that it is possible to assign costs to the actions in the execution trace depending on the context of the target system, and we consider it one of the most natural extensions of our work.

As a very recent follow-up work, Banescu and Zannone [5] have proposed to use process fitness metrics to assess the degree of compliance of an audit trial with a process model. The authors revise the definition of Levenshtein distance with the metric Φ, representing the severity of infringements. Given an element of a tentative execution sequence a and an element of a valid trace b, $\Phi(a, b)$ defines the severity of the infringement that a happened instead of b. $\Phi(a, b)$ depends on

- the reputation of the user performing the task a,
- the semantic distance between the task which is actually executed (task a) and the task defined in the process specification (task b),
- is the semantic distance between the role of the user executing the task a and the role associated to the task b in the specification, and
- the penalty due to unauthorized access to data during the execution of a task a.

The authors proposed to revise the Levenshtein distance as follows:

$$d_L^{\Phi}(\sigma, \sigma') = \begin{cases} \Phi(\cdot, b) + d_L^{\Phi}(\sigma, \sigma_b) & \text{if} \quad \sigma = \cdot \text{ and } \sigma' = b\sigma_b \\ \Phi(a, \cdot) + d_L^{\Phi}(\sigma_a, \sigma') & \text{if} \quad \sigma' = \cdot \text{ and } \sigma = a\sigma_a \\ d_L^{\Phi}(\sigma_a, \sigma'_a) & \text{if} \quad \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ \Phi(a, b) + min\{d_L^{\Phi}(\sigma_a, \sigma_b), & \text{if} \quad \sigma = a\sigma_a \text{ and} \\ d_L^{\Phi}(a\sigma_a, \sigma_b), d_L^{\Phi}(\sigma_a, b\sigma_b)\} & \qquad \sigma' = b\sigma_b \text{ and } a \neq b \end{cases} \tag{6.15}$$

This revised notion of Levenshtein distance can be one of the choices of the future extensions of our work, where the severity of infringement reflected in the distance can distinguish better between different enforcement mechanisms.

Future direction

One future direction can be the analysis of existing mechanisms to identify which one is predictable. The practically interesting question is whether an ε for predictability can be extracted from a security policy expressed as an automaton.

The second issue revolves around edit automata as an enforcement mechanism and the characterization of predictable policies. A key question is whether policies of a certain form do have always (or never) have predictable enforcement mechanism. Under some definition of convexity we could prove that convex policies always have predictable enforcement mechanisms within a bound fixed on the border, but this definition, while mathematically sound, is not intuitive enough.

6.2 Enforcement Mechanism for Error-Tolerant Policies

As we have discussed earlier, security policies describe the desired behavior of the system or the workflow. The end users of the workflow would insist on the only *one policy* describing the official protocol workflow. Hence, when an insignificant deviation from the official protocol happens, the current runtime enforcer would block the execution of such process (or a part of it in case of Iterative Suppression automatonfrom Section 5.4). As we presented in Issue 4, currently there is no generic algorithm that can construct a runtime enforcer that tolerates user errors.

In order to exit from this impasse, we build upon the works of automatic policy generation and run-time enforcement to propose a semi-automatic way to generate enforcement mechanisms that tolerate up to k errors given a texted "default" workflow and a specification of a simple list of errors and possibly their corrections.

We present our idea in Figure 6.6. Given a policy P and a maximum number of errors/deviations k, we construct a subclass of edit automata that can provably enforce the given policy by tolerating up to k errors and whose behavior is predictable (see Section 6.1).

Remark 6.2.1 *We have an aside but important note on the terminology. Most papers*

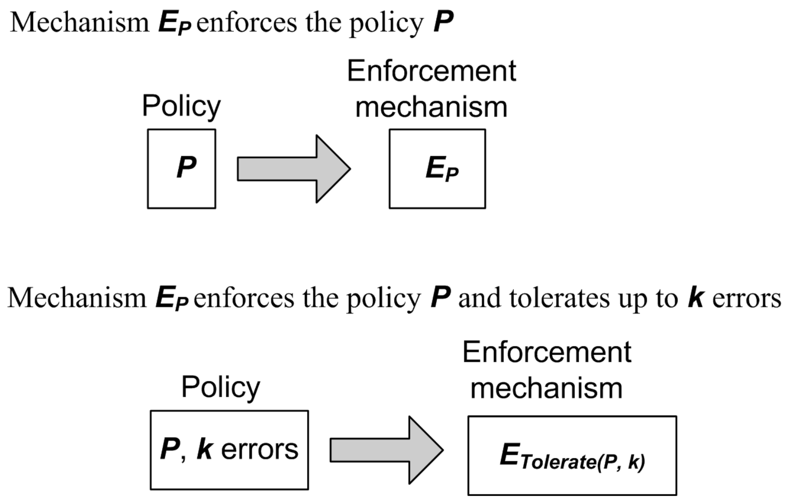

Mechanism E_P enforces the policy P
Policy
P
Enforcement mechanism
E_P
Mechanism E_P enforces the policy P and tolerates up to k errors
Policy
P, k errors
Enforcement mechanism
$E_{Tolerate(P,k)}$

Figure 6.6: Original approach and a novel approach to construction of enforcement mechanisms.

on enforcement mechanisms use the words "violation" rather than "error". We will use them interchangeably. In the course of our many interactions with Hospital San Raffaele (HSR) [1], it has become apparent that the word "error" is preferred for psychological reasons. The term "violation" implies that a doctor would deliberately ignore the steps of the protocol and this implies for the end users a deliberate mistrust in their behavior by the evil security department. Of course doctors (as any user) could misbehave but in order to gain acceptance of the mechanisms it is preferable to present enforcement as a way to support honest users rather than to deter malicious users.

6.2.1 Venial and Amendable Errors

By "error" we mean two types of deviations from the desired behavior:

- *venial* errors are not explicitly "allowed" by the policy, but they are not harmful;
- *amendable* errors are possible to be corrected at runtime.

Let us explain these notions by means of our example of drug selection process from Section 3.2. Consider an action of reviewing the therapeutical notes by the doctor. While such notes are important (as they contain information about allergies, unwanted interactions etc.) they are normally updated very rarely and for frequently used drugs doctors might "forget" to actually review them and just skip them by closing the window of drug

[1]This collaboration was done within the EU project MASTER (`www.master-fp7.eu`)

Table 6.3: A policy and possible errors

Policy	Error	Number of errors	Error type	Correction
Sequences satisfying the description of the process from running example	Instead of reviewing therapeutical notes close the window	k errors per day	Venial	No correction
	Research protocol number is not inserted		Can be corrected	Insert special number for audit

prescription instead of clicking the "Done" button. From the point of view of the medical process this can be considered a *venial error*: we can tolerate few deviations in which the logs showed that the doctor clicked "Ignore" rather than reviewed some of the most commonly prescribed drugs. More technically, a sequence of actions corresponding to this example is denoted as:

CloseNote: Dis; Tnn; Ctw; Dr; Irpn; Dpres

In this execution sequence instead of reviewing the therapeutical notes (Rtn), the doctor closes the therapeutical notes window (Ctw).

On the other hand, the doctor should not be allowed to violate the policy systematically, nor we want to overcomplicate the definition of the policy with all possible ways to treat venial errors. From a usability perspective we would just like to have a high level view, for example allowing to close the window (Ctw) instead of reviewing the therapeutical notes (Rtn) k times per day.

The enforcement mechanism we propose in this section does the rest automatically. It allows the user to make "almost" correct actions only a limited number of times and only if the errors are venial. We present Table 6.3, that can be made by an expert in the application domain. This table defines which errors can be allowed and what number of times.

The second example is inserting the research protocol number in the protocol window. A doctor has to complete this step in order to proceed with the drug selection. After she fills in this number, the reimbursement of the drug will be done by the clinical trial funds. For all the drugs that are not for research the reimbursement later on is done by the public authority as described in the File F procedure (for more details on this procedure, see Section 3.2). However, it might happen that the doctor skips the insertion of this number. Technically, this would mean that the doctor closes the protocol window (Cpw) instead of inserting research protocol number (Irpn):

CloseProt: Dis; Tnn; Rtn; Dr; Cpw; Dpres

In this case the drug reimbursement process will be done by the public authority which cannot be considered a venial error since the whole reimbursement process for this drug will be wrong (reimbursement will be done by a wrong party). The enforcement mechanism would therefore need to generate an alert and "correct" the wrong step i.e. inserting another special number that later will be used during the audit[2].

In practice, different doctors can prescribe different drugs to different patients and we would like to avoid that a local infringement of the policy (e.g. a doctor forgot to click "I have reviewed the therapeutical notes") hangs the entire process while enforcing this policy.

Notice that we need the concept of a global policy (or a "default" protocol) and we cannot spawn an enforcement mechanism for each doctor and patient pair. At first the notion of venial errors would be trivial: in the individual prescription process there is at most one venial error that could be made. Second and foremost, the hospital is liable as a whole if too many errors are present. If all doctors are allowed one venial error in the individual prescription, the hospital might end up with all processes without therapeutic notes checklist and thus the venial error would become a systematic error leading to potential lawsuits.

So we want to define how the executions where "something locally bad may happen" can be enforced by tolerating errors. The errors, which are neither venial, nor amendable, are always present and cannot be fixed in practice. For example, when the process involves the interactions of an organization with another one; for instance we can refer to the cases of outsourcing services or to the cases in which some actions are done by external parties.

In practice the policy is given implicitly by describing the workflow corresponding to legal executions (as we have shown in Example 3.2.1). Let us remind the good executions according to the drug selection process.

SimpleRun Dis; TnNn; DNr; Dpres,

NoteRun Dis; Tnn; Rtn; DNr; Dpres,

[2]In the real implementation systems are not allowed to automatically perform certain actions as the final liability must stay with a human, however they can support the human by suggesting the relevant correction.

Table 6.4: Examples of errors and their characteristics

#	Deviation e	Expected action $ex(e)$	Correction $c(e)$
1	e	a	e
2	e	a	b
3	b	a	b

Table 6.5: Almost bad traces that can be corrected

#	Trace	Expected enforcement
1	Dis; Tnn; Ctw; Dr; Irpn; Dpres	Dis; Tnn; Ctw; Dr; Irpn; Dpres
2	Dis; Tnn; Rtn; Dr; Cpw; Dpres	Dis; Tnn; Rtn; Dr; InA; Dpres
3	Dis; Tnn; Ctw; Dr; Cpw; Dpres	Dis; Tnn; Ctw; Dr; InA; Dpres

ResearchRun Dis; TnNn; Dr; Irpn; Dpres,

NoteResearchRun Dis; Tnn; Rtn; Dr; Irpn; Dpres

Recall the examples of illegal executions for this process from Example 6.1.1:

CloseNote Dis; Tnn; Ctw; Dr; Irpn; Dpres,

CloseProt Dis; Tnn; Rtn; Dr; Cpw; Dpres,

CloseNoteProt Dis; Tnn; Ctw; Dr; Cpw; Dpres

For every possible deviation/error from the legal behavior we propose the following table with the following functions. A function $ex(e)$ defines an expected action in the legal trace when an error e occurred and a function $c(e)$ defines a correction for an error e. Assuming that action e is an error and action a is an expected action of the policy, there are few alternatives summarized in Table 6.4.

Correction $c(e) = e$ (line 1) means that e is a venial error and it can be tolerated by an enforcement mechanism. A case when $c(e) = b$ (where $a \neq b$ in line 2) means that e is not a venial error, but there is a possible correction b that is by itself a venial error (line 3).

Let us come back to the example of drug selection process. Table 6.5 contains traces that can be slightly changed and then be accepted by the end users as good traces. It can be done in two ways: by allowing a venial error to occur (meaning $c(e) = e$) and by correcting it ($c(e) \neq e$). Venial error can occur instead of an action defined by the policy and is "harmless", so in our example venial error is to close the therapeutical notes window (Ctw) instead of reviewing the notes (Rtn), formally $ex(\mathsf{Ctw}) = \mathsf{Rtn}$ and $c(\mathsf{Ctw}) = \mathsf{Ctw}$. In sequence 1 of Table 6.5 the user makes this venial error and so our tolerant enforcement mechanism does not change the tentative execution of the user.

If an error is not venial, it should be corrected. We show such an example in sequence 2. Closing the protocol number window (Cpw) instead of inserting the research protocol number (Irpn) is an error that should be corrected, so we can replace this action by inserting the special number for the audit (InA), formally $ex(\mathsf{Cpw}) = \mathsf{Irpn}$, $c(\mathsf{Cpw}) = \mathsf{InA}$ and InA is a venial error. When performing the reimbursement of the drug, this number will mean that the drug is for research but the protocol number inserted will be some default number. By doing so we assure that the drug reimbursement will be done by a correct party (clinical trial funds). Sequence 3 contains both types of errors: a venial error of closing the therapeutical notes window (Ctw) instead of reviewing these notes (Rtn) and an error of closing the protocol number window (Cpw) instead of inserting the research protocol number (Irpn). Only error Cpw should be corrected because Ctw is a venial error.

We propose a quasi-metric based on the replacing distance that counts a number of venial errors.

Definition 6.2.1 *The* replacing distance with venial and amendable errors *between two finite traces is number of replacements that should be done in σ to get σ' whenever the replacement is a venial or an amendable error. Total function $d_R^{va} : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N} \cup \{\infty\}$ is a quasi-metric, such that*

$$d_R^{va}(\sigma, \sigma') = \begin{cases} 0 & \text{if } \sigma = \cdot \text{ and } \sigma' = \cdot \\ d_R^{va}(\sigma_a, \sigma'_a) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + d_R^{va}(\sigma, \sigma') & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = b\sigma_b \text{ and } a \neq b \text{ and} \\ & ex(a) = b \\ \infty & \text{otherwise} \end{cases}$$

For example, if we compare a tentative execution sequence $\mathsf{CloseNoteProt}$and a legal sequence $\mathsf{NoteResearchRun}$, we get the following replacing distance with venial and amendable errors:

$$d_R^{va}(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) =$$
$$d_R^{va}(\mathsf{Dis}; \mathsf{Tnn}; \mathsf{Ctw}; \mathsf{Dr}; \mathsf{Cpw}; \mathsf{Dpres}, \mathsf{Dis}; \mathsf{Tnn}; \mathsf{Rtn}; \mathsf{Dr}; \mathsf{Irpn}; \mathsf{Dpres}) = 2$$

because $ex(\mathsf{Ctw}) = \mathsf{Rtn}$, $ex(\mathsf{Cpw}) = \mathsf{Irpn}$, and the rest of the actions in these two sequences are equal.

We will use this distance when we compare a tentative execution sequence with the legal sequence. For comparing the outputs of an enforcement mechanism we need another distance, that calculates only venial errors:

Definition 6.2.2 *The* replacing distance with venial errors *between two finite traces is number of replacements that should be done in σ to get σ' whenever the replacement is a venial error. Total function $d_R^v : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ is a quasi-metric, such that*

$$d_R^v(\sigma, \sigma') = \begin{cases} 0 & \text{if } \sigma = \cdot \text{ and } \sigma' = \cdot \\ d_R^v(\sigma_a, \sigma'_a) & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = a\sigma'_a \\ 1 + d_R^v(\sigma, \sigma') & \text{if } \sigma = a\sigma_a \text{ and } \sigma' = b\sigma_b \text{ and } a \neq b \text{ and} \\ & ex(a) = b \text{ and } c(a) = a \\ \infty & \text{otherwise} \end{cases}$$

By E_k we denote an enforcement mechanism can tolerate up to k venial errors and correct amendable errors. For example, E_k can modify the illegal execution sequences in this way:

$$E_k(\mathsf{CloseNoteProt}) = E_k(\mathsf{Dis};\mathsf{Tnn};\mathsf{Ctw};\mathsf{Dr};\mathsf{Cpw};\mathsf{Dpres}) = \mathsf{Dis};\mathsf{Tnn};\mathsf{Ctw};\mathsf{Dr};\mathsf{InA};\mathsf{Dpres}$$
$$E_k(\mathsf{NoteResearchRun}) = \mathsf{NoteResearchRun} = \mathsf{Dis};\mathsf{Tnn};\mathsf{Rtn};\mathsf{Dr};\mathsf{Irpn};\mathsf{Dpres}$$

Let us compare these outputs of an enforcement mechanism E_k using the distance with venial errors:

$$d_R^v(\mathsf{Dis};\mathsf{Tnn};\mathsf{Ctw};\mathsf{Dr};\mathsf{InA};\mathsf{Dpres}, \mathsf{Dis};\mathsf{Tnn};\mathsf{Rtn};\mathsf{Dr};\mathsf{Irpn};\mathsf{Dpres}) = 2$$

because $ex(\mathsf{Ctw}) = \mathsf{Rtn}$ and $c(\mathsf{Ctw}) = \mathsf{Ctw}$; $ex(\mathsf{Cpw}) = \mathsf{Irpn}$ and $c(\mathsf{InA}) = \mathsf{InA}$, and the rest of the actions in these two sequences are equal.

We will use the distance d_R^v to compare the outputs of enforcement mechanisms in order to ensure that the outputs contain only venial errors, and no amendable errors.

6.2.2 Construction

In this section we present a construction of an enforcement mechanism in a form of edit automaton. The input to our construction is a default security policy P, a maximum number of venial errors k and two functions: function ex that defines an expected action for a given invalid action and function c that defines a correction of this action. We will denote these inputs by $\mathcal{I} = \langle P, k, c, ex \rangle$. The result is an edit automaton that transforms the bad executions of the system by tolerating up to k errors.

Given the policy P represented as a Policy automaton $A^P = \left\langle \Sigma, Q^P, q_0^P, \delta^P, F^P \right\rangle$, we construct an edit automaton with the states $(q, q_F)^{\sharp s,p}$, where q is a state of a Policy automaton, q_F is the last accepting state before reaching q in some run, s is a number of venial errors so far (we write s to denote the measure of soundness) and p is a number of corrections made to the original execution (we write p to denote the measure of precision). Notice that p is not bounded, it only gets increased when we correct amendable error in a tentative execution sequence and is reset when the iteration is over.

Similar to Iterative Suppression automaton from Section 5.4, the constructed automaton in this section also recognizes an execution as a sequence of iterations. The idea behind this construction is that all the good iterations are not changed and all the bad iterations are corrected if they contain up to k venial errors. Iterations that have more than k venial errors are suppressed as soon as $(k + 1)$st error arrives. The amount of venial errors so far is controlled by the state variable s.

The set of states of edit automaton for a given k is

$$Q = \{(q, q_F)^{\sharp s,p} | q \in Q^P, q_F \in F^P, 0 \leq s \leq k\} \cup \{(q_\perp, q_F)^{\sharp k,0} | q_F \in F^P\}$$

and an initial state is $q_0 = (q_0^P, q_0^P)^{\sharp 0,0}$. We define the semantics of enforcement mechanism in Figure 6.7. If a tentative execution sequence satisfies the policy, the edit automaton "copies" the Policy automaton. If the sequence is accepted, it resets the counter of errors [Good-Out]. If the sequence is not accepted, the automaton keeps counting the errors [Good-Wait].

Otherwise a tentative execution sequence does not satisfy the policy. So, we check whether more venial errors are allowed ($s < k$) and whether there exists a transition starting at the state q on the expected action $ex(e)$ that does not have to be corrected ($c(e) = e$ in [Venial-Out] and [Venial-Wait]). If such action has to be corrected, we use $c(e)$ that defines an appropriate correction (rules [Correct-Out] and [Correct-Wait]). If none of the previous cases holds, we check whether this action can initiate a new good iteration from the last visited accepting state q_F (rules [Iteration-Out] and [Iteration-Wait]). If there is no new iteration, we reach an error state (rule [Error]).

Notice that if there are no venial or amendable errors in the tentative execution, this automaton behaves as an *Iterative Suppression automaton.* The first 5 rules define the same semantics as the rules of Iterative Suppression automaton that we have defined in Figure 5.8.

Even though this construction seems to be standard, it automatically generates an

$$\textsc{Good-Out}\ \frac{q \xrightarrow{a} q' \qquad q' \in F^P}{((q, q_F)^{\sharp s,p}, a; \sigma_i, \sigma_k) \overset{\sigma_k;a}{\hookrightarrow} ((q', q')^{\sharp 0,0}, \sigma_i, \cdot)}$$

$$\textsc{Good-Wait}\ \frac{q \xrightarrow{a} q' \qquad q' \notin F^P}{((q, q_F)^{\sharp s,p}, a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F)^{\sharp s,p}, \sigma_i, \sigma_k; a)}$$

$$\textsc{Error}\ \frac{\mathit{Otherwise}}{((q, q_F)^{\sharp s,p}, e; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q_\bot, q_F)^{\sharp k,0}, \sigma_i, \cdot)}$$

$$\textsc{Iteration-Out}\ \frac{q \xrightarrow{a} \bot \qquad q_F \xrightarrow{a} q' \qquad q' \in F^P}{((q, q_F)^{\sharp s,p}, a; \sigma_i, \sigma_k) \overset{a}{\hookrightarrow} ((q', q')^{\sharp 0,0}, \sigma_i, \cdot)}$$

$$\textsc{Iteration-Wait}\ \frac{q \xrightarrow{a} \bot \qquad q_F \xrightarrow{a} q' \qquad q' \notin F^P}{((q, q_F)^{\sharp s,p}, a; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F)^{\sharp 0,0}, \sigma_i, a)}$$

$$\textsc{Venial-Out}\ \frac{q \xrightarrow{e} \bot \qquad q \xrightarrow{ex(e)} q' \qquad q' \in F^P \qquad c(e) = e \qquad s < k}{((q, q_F)^{\sharp s,p}, e; \sigma_i, \sigma_k) \overset{\sigma_k;e}{\hookrightarrow} ((q', q')^{\sharp 0,0}, \sigma_i, \cdot)}$$

$$\textsc{Venial-Wait}\ \frac{q \xrightarrow{e} \bot \qquad q \xrightarrow{ex(e)} q' \qquad q' \notin F^P \qquad c(e) = e \qquad s < k}{((q, q_F)^{\sharp s,p}, e; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F)^{\sharp s+1,p}, \sigma_i, \sigma_k; e)}$$

$$\textsc{Correct-Out}\ \frac{q \xrightarrow{e} \bot \qquad q \xrightarrow{ex(e)} q' \qquad q' \in F^P \qquad c(e) \neq e \qquad s < k}{((q, q_F)^{\sharp s,p}, e; \sigma_i, \sigma_k) \overset{\sigma_k;c(e)}{\hookrightarrow} ((q', q')^{\sharp 0,0}, \sigma_i, \cdot)}$$

$$\textsc{Correct-Wait}\ \frac{\begin{array}{c}\text{if } c(e) = ex(e) \text{ then } s' = s \text{ else } s' = s + 1 \\ q \xrightarrow{e} \bot \qquad q \xrightarrow{ex(e)} q' \qquad q' \notin F^P \qquad c(e) \neq e \qquad s < k\end{array}}{((q, q_F)^{\sharp s,p}, e; \sigma_i, \sigma_k) \overset{\cdot}{\hookrightarrow} ((q', q_F)^{\sharp s',p+1}, \sigma_i, \sigma_k; c(e))}$$

Figure 6.7: Semantics for Error-toleration mechanism constructed from the Policy automaton.

enforcement mechanism from the policy P, number of errors k and functions that define expected actions and corrections.

6.2.3 Formal Properties

Lemma 6.2.1 *Given inputs $\mathcal{I} = \langle P, k, c, ex \rangle$, an enforcement mechanism E_k with the semantics from Figure 6.7 is transparent:*

$$\forall \sigma \in \Sigma^* : \widehat{P}(\sigma) \Rightarrow E_k(\sigma) = \sigma \tag{6.16}$$

Proof. For every sequence σ that satisfies the policy P, there exists an accepting run in the policy automaton A^P. Therefore, only rules [Good-Wait] and [Good-Out] are used and the whole sequence σ is kept in the memory and is output upon reaching an accepting state of the Policy automaton. Hence, E_k is transparent. □

Lemma 6.2.2 *Given inputs $\mathcal{I} = \langle P, k, c, ex \rangle$, an enforcement mechanism E_k with the semantics from Figure 6.7 for every iteration $\sigma_P \in P$ is such that*

$$\forall \sigma \in \Sigma^* : (d_R^{va}(\sigma, \sigma_P) \leq k \Rightarrow d_R^{v}(E_k(\sigma), E_k(\sigma_P)) \leq k) \tag{6.17}$$

Proof. From the definition of replacing distance with venial and amendable errors it follows that $d_R^{va}(\sigma, \sigma_P) \leq k$ means $|\sigma| = |\sigma_P|$ and among all indices $0 < i \leq |\sigma|$ there are up to k indices $i_1, ... i_n$ $(n \leq k)$ such that for all $0 < j \leq n$: $\sigma[i_j] \neq \sigma_P[i_j]$ and $ex(\sigma[i_j]) = \sigma_P[i_j]$. For all the other indices l, we have $\sigma[l] = \sigma_P[l]$.

Therefore, for the actions $\sigma[i_1], ..., \sigma[i_n]$ one of the rules [Venial-Out], [Venial-Wait], [Correct-Out] or [Correct-Wait] holds. Since $ex(\sigma[i_j]) = \sigma_P[i_j]$, there is always a transition on $ex(\sigma[i_j])$ in the Policy automaton and it's always true that $s < k$ since there are only up to k actions in σ that are different from σ_P.

For all the other indices l different from $i_1, ..., i_n$ we have $\sigma[l] = \sigma_P[l]$. Hence, only rules [Good-Out] or [Good-Wait] will be used.

So all the rules, where the action of σ is transformed to another action, will be applied up to k times. Therefore, for every iteration σ_P (that by definition brings a Policy automaton to an accepting state) every sequence σ such that $d_R^{va}(\sigma, \sigma_P) \leq k$ will be transformed to a sequence $E_k(\sigma)$, such that $d_R^{v}(\sigma, E_k(\sigma_P)) \leq k$. Then, according to Lemma 6.2.1, $E_k(\sigma_P) = \sigma_P$, therefore theorem is proven. □

Theorem 6.2.1 *Given inputs $\mathcal{I} = \langle P, k, c, ex \rangle$, an enforcement mechanism E_k with the semantics from Figure 6.7*

Table 6.6: Errors and their characteristics for drug dispensation process

Deviation e	Expected action $ex(e)$	Correction $c(e)$
The doctor does not review the therapeutical notes (Ctw)	The doctor reviews therapeutical notes (Rtn)	The doctor does not review the therapeutical notes (Ctw)
The doctor does not insert the research protocol number for the research drug (Cpw)	The doctor inserts the research protocol number (Irpn)	Insert a special number for auditing (InA)

- *is transparent:* $\forall \sigma \in \Sigma^* : \widehat{P}(\sigma) \Rightarrow E_k(\sigma) = \sigma$,
- *is predictable within k – for every iteration $\sigma_P \in P$:*

$$\forall \nu \geq k : \exists \delta > 0 : \forall \sigma \in \Sigma^* : (d_R^{va}(\sigma, \sigma_P) \leq \delta \Rightarrow d_R^{v}(E_k(\sigma), E_k(\sigma_P)) \leq \nu) \qquad (6.18)$$

Proof. Given enforcement mechanism E_k is transparent according to Lemma 6.2.1, it is predictable since for every $\nu \geq k$ there exists $\delta = k$ such that

$$\forall \sigma \in \Sigma^* : (d_R^{va}(\sigma, \sigma_P) \leq k \Rightarrow d_R^{v}(E_k(\sigma), E_k(\sigma_P)) \leq k \leq \nu)$$

according to Lemma 6.2.2 □

6.2.4 Enforcement of Drug Selection Process while Tolerating Errors

Suppose $k = 2$ and the policy P represents a "default" protocol for a drug selection process from Section 3.2. The functions for expected actions and corrected actions are shown in Table 6.6.

Let us show how the illegal execution sequences for this process are modified by an enforcement mechanism E_k constructed following the semantics from Figure 6.7.

The first sequence CloseNoteProt= Dis; Tnn; Ctw; Dr; Cpw; Dpres contains one venial error: closing window action Ctw instead of reviewing therapeutical notes Rtn; and one amendable error: closing a protocol window Cpw instead of inserting the research protocol number Irpn.

When we compare this tentative execution sequence and a valid sequence $\mathsf{NoteResearchRun} = \mathsf{Dis}; \mathsf{Tnn}; \mathsf{Rtn}; \mathsf{Dr}; \mathsf{Irpn}; \mathsf{Dpres}$, we have

$$d_R^{va}(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$$

since $ex(\mathsf{Ctw}) = \mathsf{Rtn}$ and $ex(\mathsf{Cpw}) = \mathsf{Irpn}$.

An enforcement mechanism E_k tolerates the venial error and corrects an amendable error, so it modifies the sequence $\mathsf{CloseNoteProt}$ as follows:

$$E_k(\mathsf{CloseNoteProt}) = E_k(\mathsf{Dis}; \mathsf{Tnn}; \mathsf{Ctw}; \mathsf{Dr}; \mathsf{Cpw}; \mathsf{Dpres}) = \mathsf{Dis}; \mathsf{Tnn}; \mathsf{Ctw}; \mathsf{Dr}; \mathsf{InA}; \mathsf{Dpres}$$

Now let us find the distance between the outputs of E_k given these two sequences:

$$d_R^{v}(E_k(\mathsf{CloseNoteProt}), E_k(\mathsf{NoteResearchRun})) =$$
$$d_R^{v}(\mathsf{Dis}; \mathsf{Tnn}; \mathsf{Ctw}; \mathsf{Dr}; \mathsf{InA}; \mathsf{Dpres}, \mathsf{Dis}; \mathsf{Tnn}; \mathsf{Rtn}; \mathsf{Dr}; \mathsf{Irpn}; \mathsf{Dpres}) = 2$$

since $ex(\mathsf{Ctw}) = \mathsf{Rtn}$, $c(\mathsf{Ctw}) = \mathsf{Ctw}$, $ex(\mathsf{InA}) = \mathsf{Irpn}$ and $c(\mathsf{InA}) = \mathsf{InA}$.

Notice that the behavior of E_k described in the example above corresponds to the expected behavior that we described in Table 6.5.

6.2.5 Enforcement of an Anonymization Policy

Here we discuss one possible enforcement of an anonymization policy introduced in Example 5.4.2. Recall that this policy allows only anonymized personally identifiable data (pid) leave the system (corresponding pattern SendA^*).

In Section 5.4.5 we noticed that there is one more way to enforce anonymization policy except for those provided in that chapter. We noticed that whenever a system attempts to send non-anonymized pid, the enforcement mechanism can potentially substitute this event to anonymization action followed by sending of previously anonymized data.

This modification corresponds to a case of amendable error: sending a non-anonymized pid event SendN is an error, that is not allowed by the policy, however, it can be amended by substituting SendN to a sequence of events $\mathsf{Anonym}; \mathsf{SendA}$. We can define a correction function $c(\mathsf{SendN}) = \mathsf{Anonym}; \mathsf{SendA}$, and use our Error-toleration mechanism automatically constructed from a security policy SendA^*. For an action SendN the expected action will be defined by a function $ex(\mathsf{SendN}) = \mathsf{SendA}$, and hence all the actions SendN will be substituted by $\mathsf{Anonym}; \mathsf{SendA}$ according to the rule CORRECT-OUT from Figure 6.7.

6.2.6 Discussion

In this section we used one assumption that k *exists*. We have made such conclusion after several discussions with our partners from Hospital San Raffaele (HSR) in Milan, Italy. According to these discussions, k exists, but it is not a single number for all the business processes in the hospital, all the doctors and for all the years. In fact, it depends on a number of parameters. Our discussions with HSR leads to a conclusion that the head of the pharmacists and risk manager can agree on a particular k. For example, k can be a number that depends on a particular doctor, a year, a particular business process, a patient and a particular drug.

Our model considers one k and it is an abstraction of the reality that we have observed. This is one of the assumptions we took based on our experience. In general case and in other case studies, other assumptions can be made and they can influence the way the enforcement of the security policies should be done. For example, in some legal system, laws may not be specified very precisely to leave room for interpretation.

Let us come back to our original assumption about existence of k. On one hand, specifying one policy and tolerating errors is easier to understand by users than the class of situations in the legal system, where users would invite experts to "translate" the meaning of this class of rules. On the other hand, specifying one policy and tolerating errors is a particular case of the class of situations.

6.3 Summary

We have made two important contributions in this chapter.

First, we have discussed how to go beyond the (only) two classical properties used to evaluate an enforcement mechanism: *soundness* and *transparency*. We have gradually defined several notions that could describe predictable behavior and checked them against the industrial case study on e-Health. The idea behind *predictability* is that there are "no surprises on bad inputs". In this way we have addressed Issue 3.

Second, we have addressed Issue 4 by constructing in a semi-automatic way an enforcement mechanism that can tolerate up to k errors given a "default" policy and a specification of a simple list of errors and possibly their corrections.

We distinguished between two types of errors: venial and amendable errors. The first group are simply not critical while the second group represents errors that can be corrected. There is always yet another kind of actions called observable actions, that cannot be changed but only observed. In order to avoid what auditors call 'systematic errors" we limit the number of errors that the enforcement mechanism can tolerate.

Chapter 7

Conclusions

This thesis has addressed several problems of runtime enforcement theory, starting from the gap between the security policies one can write and the enforcement mechanisms built for them, finishing with constructing enforcement mechanisms that can tolerate user errors. In this final chapter of the thesis we summarize our primary contributions that improve the state of the art in runtime enforcement theory.

We have presented a number of challenges in runtime enforcement theory in Section 1.2. This section shows how all these challenges have been met by our contributions.

Classification of enforcement mechanisms We addressed the first challenge of this thesis (Issue 1) by investigating relations between different kinds of edit automata. As a result, we have discovered that what makes them different is not what they do when an execution is valid (because nothing should happen according to transparency), but what actually happens when an execution is invalid. The results of this contribution have been published in [12, 14].

Automatic construction of enforcement mechanisms and iterative properties To the best of our knowledge, the state-of-the-art literature does not propose an automatic construction of an enforcement mechanism from a given security policy (Issue 2). In this thesis we provided several algorithms to construct an edit automaton from a security policy represented as a Policy automaton (an automaton that combines the acceptance conditions of a finite-state automaton and a Büchi automaton). We first followed the idea of a construction found in the proof of Theorem 8 from [7], as a result we obtained a particular kind of edit automata, that we call *Longest-valid-prefix automa-*

ton, because it always outputs the longest valid prefix of the given tentative execution sequence.

We also propose a construction of another particular kind of edit automaton that is able to enforce a special kind of security properties in a more acceptable way than it is done by a Longest-valid-prefix automaton. This new mechanism is called *Iterative Suppression automaton*, it suppresses illegal parts of the invalid execution.

We have further investigated the class of policies enforceable by Iterative Suppression automata and found that they are capable of enforcing a new class of properties called *iterative properties*. We have also established a relation between iterative properties and safety, liveness and renewal properties.

We showed the difference between the Longest-valid-prefix automaton and the Iterative Suppression automaton by means of examples, representing simple policies and a policy from our industrial case study. The results of this contribution have been published in [17, 15].

Predictability We have found that there is a gap in the theory of runtime enforcement: two enforcement mechanisms can effectively enforce (i.e., while being sound and transparent) a given security policy in completely different ways. In Issue 3 we have stated that the notions of soundness and transparency are not sufficient.

We have proposed a new notion that characterizes the behavior of enforcement mechanisms when they transform bad executions. This notion called *predictability* complements the notions of soundness and transparency. We have presented several examples of predictable and not predictable enforcement mechanisms for our main case studies in this thesis. The results of this contribution have been published in [16].

Error-toleration We have addressed Issue 4 by constructing in a semi-automatic way an enforcement mechanism that can tolerate up to k errors. Given a "default" policy, that describes a predefined behavior of the system, and a specification of a simple list of errors with their possible corrections, we propose a novel construction of edit automata.

We have proven the formal properties of this kind of edit automata and explained how this enforcement mechanism can enforce a "default" policy from our industrial case study. The results of this contribution have been published in [13].

In summary, solutions to all the problems specified in this thesis are given. By the main contributions indicated above, we have presented a number of improvements upon the state of the art. Our contribution on the classification of enforcement mechanisms explains the gap in the theory of runtime enforcement; our construction of enforcement mechanisms and error-toleration provides a new approach to make the theory of runtime enforcement more constructive; finally, our notion of predictability proposes to make the theory of runtime enforcement more predictable.

Bibliography

[1] I. Aktug and K. Naliuka. Conspec – a formal language for policy specification. In *Proceedings of the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems*, volume 197 of *Electronic Notes in Theoretical Computer Science*, pages 45–58. Elsevier Science, 2007.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[4] A.V. Arkhangel'skii and L.S. Pontryagin. *General topology I : basic concepts and constructions, dimension theory*. Springer-Verlag, 1990.

[5] Sebastian Banescu and Nicola Zannone. Measuring privacy compliance with process specications. In *Proceedings of the 3rd. International Workshop on Security Measurements and Metrics*, 2011. To appear.

[6] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, pages 95–104. DIKU Technical report, 2002.

[7] 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.

[8] K. J. Biba. Integrity considerations for secure computer systems. *MITRE Co., technical report ESD-TR 76-372*, 1977.

[9] N. Bielova, M. Dalla Torre, N. Dragoni, and I. S. R. Siahaan. Matching policies with security claims of mobile applications. In *Proceedings of the Third International Conference on Availability, Reliability and Security*, pages 128–135. IEEE Computer Society Press, 2008.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*, 2011. To appear.

[11] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming*, 78(5):340–358, 2009.

[12] N. Bielova and F. Massacci. Do you really mean what you actually enforced? In *Proceedings of the 5th International Workshop on Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag Heidelberg, 2008.

[13] N. Bielova and F. Massacci. Computer-aided generation of enforcement mechanisms for error-tolerant policies. In *Proceedings of the International Symposium on Policies for Distributed Systems and Networks (POLICY'11)*, pages 89–96. IEEE Computer Society Press, 2011.

[14] N. Bielova and F. Massacci. Do you really mean what you actually enforced? *International Journal of Information Security*, pages 239–254, 2011. 10.1007/s10207-011-0137-2.

[15] N. Bielova and F. Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*, 2011. To appear.

[16] N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, volume 6542 of *Lecture Notes in Computer Science*, pages 73–86. Springer-Verlag, 2011.

[17] N. Bielova, F. Massacci, and A. Micheletti. Towards practical enforcement theories. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag Heidelberg, 2009.

[18] N. Bielova and I. Siahaan. Testing decision procedures for security-by-contract. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[19] A. Bohannon and B. C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *Proceedings of the USENIX Conference on Web Application Development 2010*, 2010. To be published.

[20] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Communications and Computer Security*, pages 79–90. ACM, 2009.

[21] D. F.C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, 1989.

[22] A. Brown and M. Ryan. Synthesising monitors from high-level policies for the safe execution of untrusted software. In *Proceedings of the 4th Information Security Practice and Experience Conference*, Lecture Notes in Computer Science, pages 233–247. Springer-Verlag Heidelberg, 2008.

[23] H. Chabot, R. Khoury, and N. Tawbi. Generating in-line monitors for rabin automata. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 2009.

[24] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, Lecture Notes in Computer Science, pages 474–486. Springer-Verlag, 1992.

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

[26] David A. Chappell. *Enterprise Service Bus.* O'Reilly Media, 2004.

[27] K. Chatterjee, L. Doyen, and T. A. Henzinger. Expressiveness and closure properties for quantitative languages. *Computing Research Repository*, abs/1007.4018, 2010.

[28] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proceedings of the 2008 IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[29] D.L. Cohn. *Measure Theory.* Birkhauser, 1980.

[30] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE Computer Society Press, 2010.

[31] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Communications and Computer Security*, pages 38–48. ACM Press, 1998.

[32] Zach Epstein. Featured facebook stole every contact and phone number in your phone heres how to undo the damage. `http://www.bgr.com/2011/08/12/facebook`, August 2011.

[33] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, 2003.

[34] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Proceedings of the Fourth International Conference on Information Systems Security (ICISS'08)*, Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2008.

[35] Y. Falcone, J.-C. Fernandez, and L. Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *Proceedings of the 24th ACM Symposium on Applied Computing – Software Verification and Testing Track (SAC-SVT)*, pages 593–600. ACM Press, 2009.

[36] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Proceedings of the 9th International Workshop on Runtime Verification (RV'09)*, Lecture Notes in Computer Science, pages 40–59, Heidelberg, 2009. Springer-Verlag.

[37] Jan Fanta, Petr Svojanovsky, Ivana Sabatova, Klaus Julisch, Emmanuel Pigout, Claire Worledge, and Andrea Micheletti. Stakeholder requirements analysis. Public Deliverable of EU Research Project D1.1.2, MASTER- Managing Assurance, Security and Trust for Services, Report available at www.master-fp7.eu, 2009.

[38] P.W.L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society Press, 2004.

[39] Cdric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. In *ACM Transactions on Programming Languages and Systems*, pages 307–318. ACM Press, 2001.

[40] G. Gheorghe, S. Neuhaus, and B. Crispo. xESB: An enterprise service bus for access and usage control policy enforcement. In *Proceedings of the fourth IFIP WG 11.11 International Conference on Trust Management*, volume 321, pages 63–78. Springer Boston, 2010.

[41] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[42] Dieter Gollmann. *Computer security (2. ed.)*. Wiley, 2005.

[43] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[44] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference (ASIAN'06)*, volume 4435, pages 75–89. Springer-Verlag Heidelberg, 2006.

[45] 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.

[46] R. Khoury and N. Tawbi. Corrective enforcement of security policies. In *Proceedings of the 7th International Workshop on Formal Aspects in Security and Trust*, Lecture Notes in Computer Science, pages 176–190. Springer-Verlag, 2010.

[47] R. Khoury and N. Tawbi. Using Equivalence Relations for Corrective Enforcement of Security Policies. In *Proceedings of the 5th International Conference, on Mathematical Methods, Models, and Architectures for Computer Network Security*, volume 6258 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2010.

[48] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Computational analysis of run-time monitoring: Fundamentals of java-mac. *Electronic Notes in Theoretical Computer Science*, 70(4):80 – 94, 2002. Proceedings of the International Workshop on Runtime Verification (FLoC Satellite Event).

[49] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[50] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. An English translation of the "Physics Sections" of the Proceedings of the Academy of Sciences of the USSR.

[51] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[52] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European Symposium on Research in Computer Security*, volume 6345 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag Heidelberg, 2010.

[53] D. Marino, J.-J. Portal, M. Hall, C. Bastos Rodriguez, P. Soria Rodriguez, J. Sobota, J. Miksu, and Y. Dwi Wardhana Asnar. Master scenarios. Public Deliverable of EU Research Project D1.2.1, MASTER- Managing Assurance, Security and Trust for Services, Report available at www.master-fp7.eu, 2008.

[54] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. In *Proceedings of the Second International Workshop on Security and Trust Management*, volume 179 of *Electronic Notes in Theoretical Computer Science*, pages 31–46. Elsevier Science Publishers B.V., 2007.

[55] I. Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electronic Notes in Theoretical Computer Science*, 186:101–120, 2007.

[56] David W. Mount. *Bioinformatics: Sequence and Genome Analysis, Second Edition.* Cold Spring Harbor Laboratory Press, 2nd edition, 2004.

[57] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for usage control. In *Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 240–244. ACM Press, 2008.

[58] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

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

[60] 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.

[61] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2-4):158–184, 2007.

[62] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems.* PhD thesis, University of Pennsylvania, 2000.

[63] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[64] D. Yun, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–249. ACM Press, 2007.