Università di Pisa

Ph.D. Thesis

On the Security of Software Systems and Services

Gabriele Costa

Supervisor
Pierpaolo Degano

Supervisor
Fabio Martinelli

November 21, 2011

Abstract

This work investigates new methods for facing the security issues and threats arising from the composition of software. This task has been carried out through the formal modelling of both the software composition scenarios and the security properties, i.e., policies, to be guaranteed.

Our research moves across three different modalities of software composition which are of main interest for some of the most sensitive aspects of the modern information society. They are *mobile applications*, *trust-based composition* and *service orchestration*.

Mobile applications are programs designed for being deployable on remote platforms. Basically, they are the main channel for the distribution and commercialisation of software for mobile devices, e.g., smart phones and tablets. Here we study the security threats that affect the application providers and the hosting platforms. In particular, we present a programming framework for the development of applications with a static and dynamic security support. Also, we implemented an enforcement mechanism for applying fine-grained security controls on the execution of possibly malicious applications.

In addition to security, *trust* represents a pragmatic and intuitive way for managing the interactions among systems. Currently, trust is one of the main factors that human beings keep into account when deciding whether to accept a transaction or not. In our work we investigate the possibility of defining a fully integrated environment for security policies and trust including a runtime monitor.

Finally, *Service-Oriented Computing* (SOC) is the leading technology for business applications distributed over a network. The security issues related to the service networks are many and multi-faceted. We mainly deal with the static verification of secure composition plans of web services. Moreover, we introduce the synthesis of dynamic security checks for protecting the services against illegal invocations.

To Alberto

With or without religion, good people can behave well and bad people can do evil;
but for good people to do evil–that takes religion.

Steven Weinberg

Acknowledgments

I would like to thank my supervisors Pierpaolo Degano and Fabio Martinelli for continuously stimulating me to work better and harder (respectively!). Several colleagues and friends have contributed to the work presented in this thesis with their comments and suggestions. Among the others, I am in debt with three persons. Massimo Bartoletti introduced me to the investigation through formal methods and he is still the archetype of the researcher in my mind. Ilaria Matteucci and Paolo Mori patiently tolerated my annoying enthusiasm and my attitude to discuss and debate on pointless details. They are priceless professionals and human beings.

Many people supported me in so many ways that I cannot even remember. They are just too many to be all listed here and I just mention few of them. Donatella and Laura, who really contributed to make my dreams come true, Letizia, Silvana, Vinicio, Marco and Sonia. Finally, I thank Francesca for taking part in all our adventures, including this one.

I would like to thank Gavin Lowe, Frank Piessens, Antonio Brogi and Fabio Gadducci for their insightful comments which significantly improved the overall quality of the present work.

Part of the work presented in this thesis have been done in the scope of EU-funded project Nessos, EU-funded project Aniketos and EU-funded project Connect.

Contents

Introduction **3**
1.1 Motivations and techniques . . . 3
1.2 Goal and structure of this thesis . . . 4

Background **9**
2.1 Security models . . . 9
2.1.1 Access Control . . . 9
2.1.2 Usage Control . . . 12
2.1.3 History-based security . . . 13
2.2 Security analysis mechanisms . . . 14
2.2.1 Type systems . . . 14
2.2.2 Model checking . . . 16
2.3 Trace properties specification . . . 18
2.3.1 Automata-based Specification . . . 18
2.3.2 Process Algebras . . . 22
2.3.3 Modal and temporal logics . . . 23
2.4 Composed systems security in the field . . . 24
2.4.1 J2ME security . . . 25
2.4.2 Web services and grid computing . . . 26

Mobile Application Security and Enforcement on Devices **29**
3.1 Extending Java with local policies . . . 30
3.1.1 Local policies specification . . . 31
3.1.2 Policy sandbox . . . 38
3.1.3 Security checks deployment . . . 40
3.2 The Jalapa framework . . . 43
3.2.1 Framework structure . . . 43
3.2.2 The Jisel runtime environment . . . 45
3.2.3 Static analysis and verification . . . 47
3.3 On-device monitor inlining . . . 50
3.3.1 Application monitoring on mobile devices . . . 50
3.3.2 Bytecode in-lining . . . 53
3.3.3 System implementation . . . 58

3.4 A centralised monitoring architecture for mobile devices 63
3.4.1 Platform monitoring 63
3.4.2 Extensible monitoring architecture 65
3.4.3 Parental control: a case study 69
3.5 Discussion 72

Trust-Driven Secure Composition **73**
4.1 Security-by-Contract-with-Trust 73
4.1.1 Security-by-Contract Paradigm 74
4.1.2 Extending S×C with Trust 75
4.1.3 Trust management 79
4.2 Introducing Gate Automata 80
4.2.1 Gate automata 81
4.2.2 Automata semantics 84
4.2.3 Trace validity 87
4.3 S×C×T through gate automata 92
4.3.1 Gate Automata and ConSpec 92
4.3.2 Enforcement environment 95
4.4 Discussion 98

Secure Service Composition **99**
5.1 Security issues in open networks 99
5.1.1 Open networks 100
5.1.2 Service structure 104
5.1.3 Type and effect system 112
5.1.4 Typing relation 113
5.2 Modular plans for secure service composition 119
5.2.1 Properties of Plans 119
5.2.2 The "Buy Something" Case Study 122
5.3 Synthesizing security prerequisites 131
5.3.1 Security Prerequisite 131
5.3.2 Partial evaluation of policies 133
5.3.3 A strategy for service orchestration 136
5.4 Discussion 137

Conclusions **139**

Bibliography **145**

Appendix **161**

List of Figures

2.1 The UCON model. 12
2.2 A simple transition system. 14
2.3 The information flow model. 15
2.4 A Kripke structure and the visit of a model checker. 17
2.5 A usage automaton for a multi-task environment. 20
2.6 An interface automaton for the TCP protocol. 21

3.7 File confinement policy `file-confine(f,d)`. 35
3.8 Application development in Jalapa. 44
3.9 The event graph producing the history expression $H_{\texttt{log}}$. 49
3.10 Runtime monitoring architecture using a customised JVM. 54
3.11 Runtime monitoring architecture using the in-lining approach. 55
3.12 The instrumentation step of a MIDlet. 56
3.13 The in-lining of a single API call in a bytecode sequence 57
3.14 Monitoring overhead performances comparison. 61
3.15 Battery power consumption without and with a running PDP. 63
3.16 Power consumption for loading a local html page. 63
3.17 A schematic representation of the system managing three modules. . 67
3.18 Control screens for phone calls (left) and messages (right). 71

4.19 The Security-by-Contract process. 75
4.20 The extended Security-by-Contract application workflow. 77
4.21 The contract monitoring configurations. 78
4.22 A gate automaton for file access. 81
4.23 A gate automaton for the *Chinese Wall* policy. 83
4.24 A gate automaton for the *ask user* policy. 83
4.25 The instantiation of the file access gate automaton. 86
4.26 A *file closing* policy. 88
4.27 The instantiation of the *file closing* policy. 91
4.28 The ConSpec preamble, security state (left) and clauses (right). . . 93
4.29 The conversion of a ConSpec specification into a gate automaton. . . 95
4.30 The enforcement environment based on gate automata. 97

5.31 A travel booking network 103

5.32 A trivial usage automaton (self-loops are omitted). 106
5.33 Security policies as usage automata. 106
5.34 A policy saying "never two actions α on the same resource". 118
5.35 Safe plans for the travel booking network 122
5.36 The "buy something" scenario. 123
5.37 Usage automata. 125
5.38 Services deployment. 126
5.39 A valid plan rooted in service *buy*. 130
5.40 A policy rejecting unregistered clients. 132
5.41 Prerequisite policy returned by $\texttt{Preq}(H, A_\psi)$ 135

List of Tables

2.1 An instance of access matrix for two users and three files. 10
2.2 An AM for the mask `rwxr-x--x`. 11

3.3 Enforcement mechanism for policies defined through usage automata. 37
3.4 An example of ConSpec security policy. 52

4.5 The trust feedback generated by an application. 80
4.6 Gate automata instantiation. 85

5.7 Definition of Usage Automaton. 105
5.8 The syntax of λ^{req}. 107
5.9 The syntax of guards. 108
5.10 The operational semantics of λ^{req}. 110
5.11 The syntax of history expressions. 112
5.12 The semantics of history expressions. 113
5.13 Types, type environment and typing relation. 114
5.14 The flattening operator. 120
5.15 A possible implementation of *buy*, *pay* and *certify*. 127
5.16 The prerequisite synthesis algorithm 134

Introduction

Security has always been a central issue in all the stages and manifestations of the human society. The contemporary *digital society* is not an exception. Nowadays, computing systems pervade many aspect of our life and we need to provide them with appropriate mechanisms for facing possible security threats.

In a very general sense, security means guaranteeing that the rules of a system are respected. Hence, reasoning about security requires a precise understanding of the entities that are involved in the process. In this work we focus on the security of *composed systems*, i.e., the systems arising from the cooperation of many participants. Here, we use *formal methods* for the investigation and enforcement of the security properties of these systems.

1.1 Motivations and techniques

The interest in the security of software and hardware systems has been continuously increasing in the last decades. Basically, it grew along with the presence of the computing devices and the responsibilities we delegate them. Also, many of the concerns of the experts about the weakness of several systems have been confirmed by successful attacks. Many authors investigated the techniques and countermeasures to cope with the existing and potential security flaws and the growing interest in this topic is also demonstrated by the increasing number of proposals (e.g., see [84] for a survey on program verification and static analysis tools).

Among the sources of security threats, *mobility* and *compositionality* have been often considered of paramount importance. For instance, consider a monolithic system having a stable location and being only accessible to a certain category of users, i.e., the administrators. Even though such a system can still need protection mechanisms, the "degrees of freedom" of the security issues are limited to the behaviour of two types of entities, i.e., the system and its users.

However, the modern trends in programming and architectural paradigms seem to move in the opposite direction, i.e., from stable, centralised systems, to mobile, distributed ones. Mobile and compositional systems are open by design, i.e., an attacker can be a legal participant of the system. Often, there is no way to distinguish an attacker from a normal component without knowing its actual behaviour.

Composition itself is a multi-faceted concept needing a precise characterisation.

We restrict ourselves to a well defined category of composed systems. Indeed, here we consider general-purpose systems rather that entities having a specific goal. This means that we make no assumptions on the structure of the entities involved in compositions and we do not consider networks with a specific purpose, e.g., sensors networks or radio frequency identification (RFID) devices. Also, we focus on the security of compositions at the application level. This view comprises both the interaction between applications running on the same device and applications running on distributed platforms connected through a network.

Discussing the properties – and especially the security ones – of a system needs to be based on some suitable description of the system and the properties, as well. *Formal methods* can be suitably exploited for precisely defining the systems and their properties. Also and more importantly, formal methods embody the rigorousness of the mathematical reasoning.

The formal description of the computing agents offers a number of advantages. Mainly, it creates a mathematical model having well-defined axioms and transformation rules. Moreover, it is useful for modelling quite complex objects in a very compact way. If this is done through a formal language, we can focus on certain aspects of the computation that are relevant for the properties under investigation hiding irrelevant details. Similarly, the security properties of programs can be specified and processed using appropriate formalisms. A formal proof that a model satisfies a property represents the finest type of guarantee that one can have.

1.2 Goal and structure of this thesis

Our objective is presenting advancements and contributions resulting from our investigation on the security aspects of software composition. We progressively focussed on different aspects of the composition of software and services for constructively reasoning about security. In particular, our work has been structured in three macro-categories: *mobile applications*, *trust-based composition* and *service composition*. They represent three different scenarios in which security issues arise from the composition of two or more agents. Briefly, the results obtained in the first two field are exploitable in the third area for guaranteeing the security of service compositions under reasonable assumptions.

For the sake of presentation, we move from the simpler to the more complex scenario. Informally, the size of a system increases its complexity. In other words, we consider more complex a system obtained from the composition of a large number of participants than one obtained from few parties. Moreover, the complexity of the security analysis is affected by the reliability of the participants to the composition. This means that, for our purposes, the interaction between two components such that one trusts the other is simpler than the composition of two agents that do not trust each other.

It is also important to note that the current trend in network-based, distributed

systems consists in moving from communication protocols using simple, plain message to complex data structures and even mobile code. Most of the proposed standards for web service description and interaction are based on XML, e.g., see [12, 53, 41, 40, 11]. Also, a relevant part of the web content is now represented by dynamic objects, e.g., browser-interpreted functions, implemented with some scripting language, e.g., JavaScript. Moreover, emerging technologies, e.g., Java OSGi [144], use pieces of software that are packaged, delivered and composed by the participants to a service network.

According to these considerations, the simpler model of interest is represented by a platform that imports, i.e., installs and runs, an external application. We can evaluate the security aspects of this kind of composition from two different points of view: (i) the applications need protection against malicious (parts of) platforms and (ii) the platform needs protection against malicious applications. For the first aspect, we proposed an extension of the Java development framework with a special support for the verification and monitoring of *local security policies* [24] that the software designers can include in their programs. Instead, in order to deal with the second issue, we proposed a monitoring environment that provides security guarantees to a hosting platform.

A step further consists in investigating the security properties that can be provided to agents, e.g. programs or devices, that have a limited observability on each other. This may happen for several reasons, e.g., we have two programs running on two remote platforms communicating through the network. In many social interactions, the evaluation of suitable side conditions can compensate for the shortage of formal security guarantees. Also, in automated systems many security critical decisions are driven by this kind of considerations. In general, we can call *trust* the expectation of an agent that a certain interaction will successfully take place even though a risk of failure exists. Here, we present a proposal for the integration of the trust-based decisions in the framework originally introduced in [81] for providing formal security guarantees. Part of this model also consists of a runtime framework relying on a new class of automata which has been introduced for defining security and trust policies.

The third and last model is obtained by considering the composition of many, remotely deployed pieces of software. Under these assumptions we have many computational nodes laying on a structured network that communicate though predefined protocols and that can even exchange and execute remote code. These are the typical conditions for some of the distributed computing paradigms, e.g., *web services* and *cloud computing*, which are currently receiving major attention by academia and industries. We present an extension of the model originally introduced by Bartoletti et al. [23] for dealing with the security properties of the *open networks*. Moreover, we investigate the possibility of generalising the standard, call-by-contract invocation model with new security specifications, namely the *security prerequisites*. Also, in this section we show that the security guarantees, obtained through the application of the techniques presented in this thesis, are preserved through the arbitrary

composition of web services. The *compositionality of security* is one of the best properties we can ask to a security model for web services. Indeed, it guarantees that the scalability and compositionality of the target system are not compromised by the security mechanisms.

Summing up, the work presented in this thesis aims at addressing a problem, i.e., guaranteeing the security of composed systems, which is both theoretically and practically interesting. Moreover, our investigation is aligned with the current technological trends, e.g., Software-as-a-Service (SaaS). The result of our study is the definition of an approach to the security of distributed systems which is general enough to be reasonably applicable to most compositional contexts.

This thesis is organised as follows:

- **Chapter 2** presents a survey of some of the existing approaches and techniques for dealing with the security of computer systems. We focus on four different perspectives which are relevant for a better understanding of this thesis: *security models* (Section 2.1), techniques for *static analysis* (Section 2.2), *specification of properties* (Section 2.3) and some standards for *security in the field* (Section 2.4).

- **Chapter 3** introduces our work on the security of mobile application. In particular, we investigated the possibility of creating a development framework for mobile applications based on local policies (Section 3.1) and its actual implementation (Section 3.2). Then, we discuss the design a lightweight monitor for mobile applications (Section 3.3) and the complete architecture of centralised controller for mobile devices (Section 3.4).

  Chapter 3 mainly refers to the work published in [21, 22, 66, 9].

- **Chapter 4** describes the work that we carried out on the integration between security and trust in software composition. We initially introduce our proposal for a security and trust paradigm, namely *Security-by-Contract with Trust* – S×C×T (Section 4.1). Then, we present a new specification formalism, i.e., *gate automata* (Section 4.2), and we exploit them for defining the S×C×T runtime enforcement (Section 4.3).

  The work presented in Chapter 4 mainly refer to the papers [64, 65, 68].

- **Chapter 5** shows our advancements on secure service composition and the automatic synthesis of security prerequisites. We start by introducing *open networks*, i.e., networks where one or more components are not specified at verification time (Section 5.1). Then, we describe our mechanism for the verification of *modular composition plans* (Section 5.2) and we conclude by presenting our approach to the synthesis of *security prerequisites* (Section 5.3).

  Chapter 5 contains the work published in [61, 63, 62].

- **Chapter 6** concludes this work presenting some final remarks and introducing the future directions of research that we consider to be promising for a follow up of this thesis.

Background

The main purpose of this section is to provide the reader with an overview of the existing techniques and results that are relevant for the understanding of this work.

2.1 Security models

In the last decades, many authors have been working on studying the security issues arising when a platform executes a program. However, the problem of controlling the behaviour of the executing code is very traditional. For instance, the correct access to the resources need for the computation (e.g., CPU time and memory space) is a classical problem in the design of the operating systems (see [179]). Originally, the main concern about the programs abusing the shared resources was their exhaustion and the resulting reduction in the system usability. The only reason for a program to misbehave (with respect to the usage of the platform facilities and resources) was an error.

As we delegated more and more responsibilities to the computers, they started to be appealing for security attacks. Nowadays, it is very difficult, if not even impossible, to imagine a system that does not handle any resource or information that someone could be interested in accessing in some illegitimate way. Hence, the progresses in the design of the modern operating systems also produced a relevant effort for developing an adequate security support.

2.1.1 Access Control

Access Control [167, 166] is the discipline studying the mechanisms and techniques for granting that all and only the *authorised subjects* can access some critical *resources*. Roughly, the main components of an access control system are: (i) a *security policy* and (ii) a *security mechanism*. The security policy is responsible for deciding whether a resource can be accessed by a certain subject. Instead, the security mechanism represents the physical implementation of the system mediating the access to the resources according to the policy evaluation.

The security policy provides an high-level view of the access control system. In other words, the policy represents an abstract model of the access rules. This makes easier to reason about the access control configuration of a system.

	F_1	F_2	F_3
U_1	$\left\{ \begin{array}{c} \text{read} \\ \text{write} \end{array} \right\}$	{write}	$\emptyset$
U_2	$\left\{ \begin{array}{c} \text{read} \\ \text{write} \end{array} \right\}$	$\left\{ \begin{array}{c} \text{read} \\ \text{write} \end{array} \right\}$	{exec}

Table 2.1: An instance of access matrix for two users and three files.

A traditional system for defining access control policies is through an *Access Matrix* [116] (AM). Roughly, an AM has a number of rows, i.e., one for each subject, and columns, i.e., one for each resource. The content of a cell (i, j) says whether the subject i can access the resource j. Also, the cells' content defines the *type of access* the subject is allowed to do to the resource. More formally, an AM M is a matrix such that $\forall i, j \,.\, M[i, j] = P_{i,j}$ where $P_{i,j}$ is the set of access operations that the subject (of index) i can perform on the resource (of index) j. The security policy implemented by a matrix M is a ternary predicate ϕ taking a subject s_i, a resource r_j and an access operation o and defined as follows.

$$\phi(s_i, r_j, o) \iff o \in M[i, j]$$

Table 2.1 shows a simple AM M. The meaning of the matrix in terms of access control policy is very intuitive. For instance, U_2 is allowed to read F_2 as read $\in M[2, 2] = \{\text{read, write}\}$ while U_1 cannot, i.e., read $\notin M[1, 2] = \{\text{write}\}$.

The policy represented by an AM can be changed through proper commands. Basically, each command takes some subjects, some resources and some operations and changes the content of the matrix. Standard commands allow for adding/removing an access operation in/from a cell, creating/deleting a row or column. Note that the invocation of these commands can be controlled through the matrix itself. Indeed, we can use one column of the matrix for restricting the access to its own commands.

Since AMs can be efficiently implemented, it is not surprising that many real-life systems use them (or their variants). For instance, Unix-based operating systems associate a 9-bit mask to each file for representing the access rights of the users. The mask contains three blocks composed by three bits each. Every bit of a block represents an allowed (1) or denied (0) operation (i.e., read, write and execute). The blocks define the permissions of the file owner, the file group and all the other users. The following string represents the mask of a file F that can be read, written and executed by its owner (`rwx`), read and executed by the users in its group (`r-x`) and just executed by all the others (`--x`).

```
rwxr-x--x
```

	...	F	...
U^{owner}		{read, write, exec}	
U_1^{group}		{read, exec}	
⋮		⋮	
U_k^{group}		{read, exec}	
U^{other}		{exec}	
⋮		⋮	

Table 2.2: An AM for the mask `rwxr-x--x`.

In this case, the AM results from the composition of the masks of all the files in a system. For example, the structure of a matrix complying with the previous mask would be as the one depicted in Table 2.2.

The command used to modify the existing masks is `chmod`. Basically, it switches one or more bits in a mask, i.e., it adds/removes allowed operations. Similarly, `chown` and `chgrp` change the owner and the group of a file, respectively. These commands modify the content of the AM by swapping the cells of the matrix.

Rows and columns are created and removed as well. Whenever a new file is created, it gets a default mask, i.e., the matrix is extended with a new column. The opposite happens when a file is deleted. Instead, rows are created (deleted) whenever a user is registered (cancelled).

According to this model, programs represent special entities for an AM. Indeed, they are both resources, i.e., executable files, and (agents acting on behalf of the) users. Typically, when a user starts a program, the application runs with his privileges, i.e., the accesses requests are evaluated according to the user's row. This means that the users should be always aware about the behaviour of the programs. In general, this is a quite strong assumption. Partly because rarely users are concerned with the security issues (e.g., see [152, 142]). However, the main reason is that even a superficial understanding of the behaviour of a program may require technical skills and/or advanced tools.

Another problem with the basic models for access control is the expressiveness of their policies. As a matter of fact, while understanding the meaning of the policy expressed through an AM is quite simple, starting from an informal definition of the desired policy and building a corresponding AM is not straightforward and, sometimes, even not possible. For instance, one could be interested in implementing a policy saying "the user U can access only one between the two resources R and R'". As the contents of the cells of the AM are independent, i.e., we cannot define

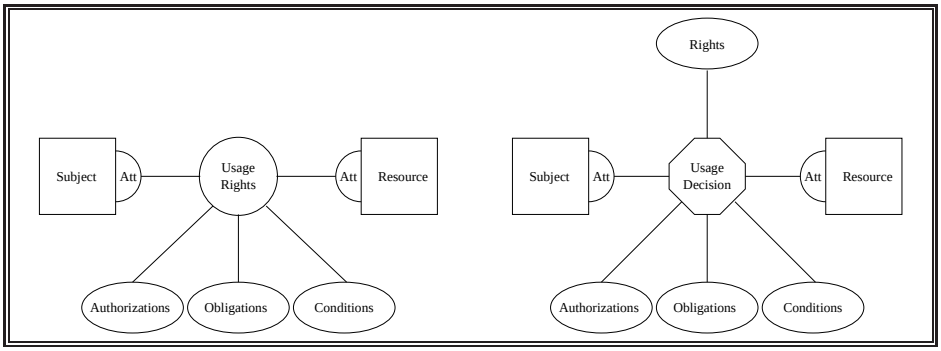


Figure 2.1: The UCON model.

relations among columns and/or rows, this policy is not expressible. Many authors proposed extensions to the original access control model in order to cope with more expressive properties. For instance, specification formalisms based on access control logics have been proposed in [186] and [101].

2.1.2 Usage Control

Recently, *usage control* received major attention from the security community (e.g., see [151, 188, 189, 158]). Usage control generalises access control by allowing for the specification of properties declaring how a subject can use a critical resource. Usage control represents a continuous and unified model for dealing with the access to the resources of a system and their usage. In particular, usage control can be adopted for modelling (and dealing with) many scenarios of interest like access control, digital right management (DRM), separation of duty (SoD) and so on.

The model presented in [151] extends access control with *Authorizations*, *oBligations* and *Conditions* (ABC). These three factors drive the *Usage Decisions*. Authorizations, obligations and conditions are defined according to the other entities in the model. They are: *rights*, i.e., the set of usage operations that a subject can invoke on a resource, and *attributes*, i.e., properties fully qualifying the subjects and resources. Some attributes can change due to the subject activity, i.e., they are *mutable*, while others never change, i.e., they are *immutable*.

Authorizations are predicates that evaluate whether, according to the involved attributes, an subject can request some rights on a resource. Obligations are predicates defining the requirements that a subject has to fulfil before a certain usage. Finally, conditions are predicates describing properties on the contextual and the environmental in which the usages takes place. The full view of the UCON components and their relations is shown in Figure 2.1.

In particular, Figure 2.1 shows the original model proposed in [151] (left) and the one presented by Zhang [188] (right). The two diagram represent the same model. Nevertheless, the model by Zhang emphasizes the usage decision process by

putting it at the center of the schema. This structure provided a reference for the design of the usage control systems. For instance, a Policy Decision Point (PDP) is a centralised agent deciding whether a certain request to use a resource is legal or not. Many implementations of the PDP include modules for the access and evaluation of the propositions dealing with authorizations, obligations and conditions.

2.1.3 History-based security

More recently *history-based security* has been a major proposal. Basically, the history-based approach to security consists in deciding which behaviours are legal or not according to the *execution history*. Execution histories represent the past behaviour of a target in terms of security-relevant operations. History-based mechanisms have been shown to be applicable for dealing with access control as well as usage control in many cases.

The current formalization of the history-based approach was proposed by Abadi and Fournet [2]. However, the idea of fully characterising the legality of the behaviour of a program after the evaluation of the states it generates is quite traditional. For instance, Alpen and Schneider [8] analyse in terms of execution history the two important categories of *safety* and *liveness* properties [114].

The history-based approach relies on a suitable representation of the history of a target. For instance, a transition system is a model of computation consisting of *states* and *transitions*. Roughly, a state is a possible configuration of the system while a transition is a move from a source state to a target one. The history of a transition system can be represented as the sequence of its transitions, the sequence of the states it visits or their alternating combination. For example, consider the transition system graphically represented in Figure 2.2. It is simple to verify that the histories 012120, abcbd and 0a1b2c1b2d0 are three possible representations for the same computation starting from state 0.

Summing up, in history-based approaches we always have (i) a model of computation using (ii) execution traces that may respect or violate a (iii) security specification.

History-based security has been proposed as a generalisation of *stack inspection*. Stack inspection is an approach to security often used by procedural programming frameworks. Indeed, during the computation these systems use a call stack to keep trace of the current sequence of invocations. Assuming that every security-relevant behaviour is performed by some precisely known procedure, the call stack can be used to decide whether the last invocation has sufficient privileges to proceed.

The main motivation to the extension of this model is that the call stack is only a fragment of the total execution history. As a matter of fact, when a procedure terminates, the corresponding entry is removed from the call stack and does not affect the security evaluation any more. However, in some cases the information about the previous method invocations can affect the system security. In particular, stack inspection cannot help when the execution of some trusted code depends on

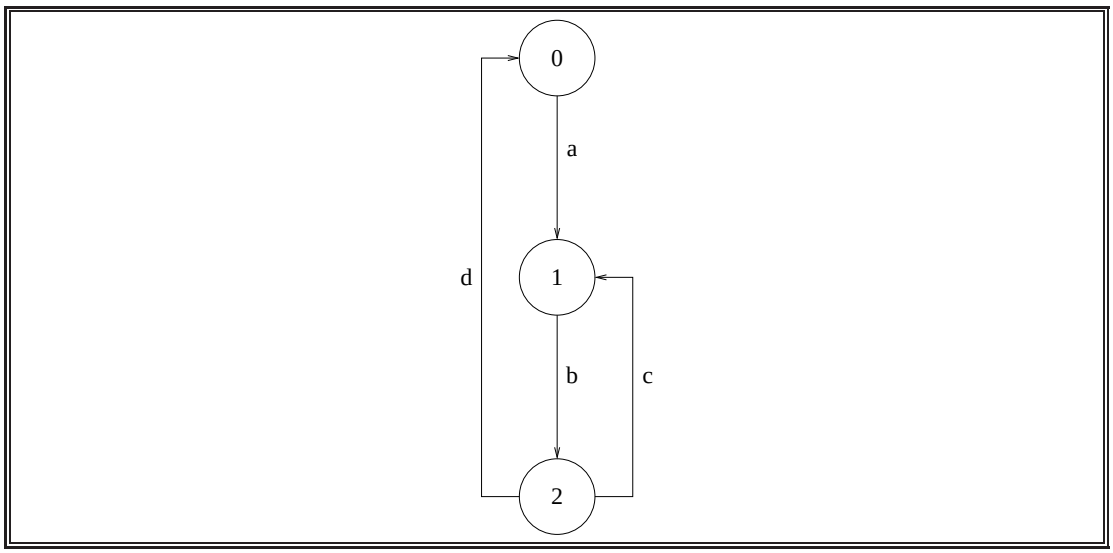


Figure 2.2: A simple transition system.

the result of untrusted methods [91].

2.2 Security analysis mechanisms

Below we present the approaches to the verification of security properties that are relevant for this work.

2.2.1 Type systems

A type system is a set of axioms and deduction rules which infer the classes of values that a program handles without executing it. Well typed programs enjoy several "desirable" properties, e.g., memory and control flow safety. Not surprisingly, many modern programming languages have strong type systems.

In general, the structure of a typing judgement is:

$$\Gamma \vdash p : \tau$$

where Γ is a type environment, p is the typed program and τ is its type. The type environment contains the type assertions that are collected and used during the typing process. Often, type assertions are simply the result of a previous typing, i.e., bindings between a part of the program and its type.

Type axioms are simple, direct type judgements. Typically they apply to some atomic instructions or expressions having a base type. For instance, constants can be easily associated to their domain as in the following case (note that here Γ is

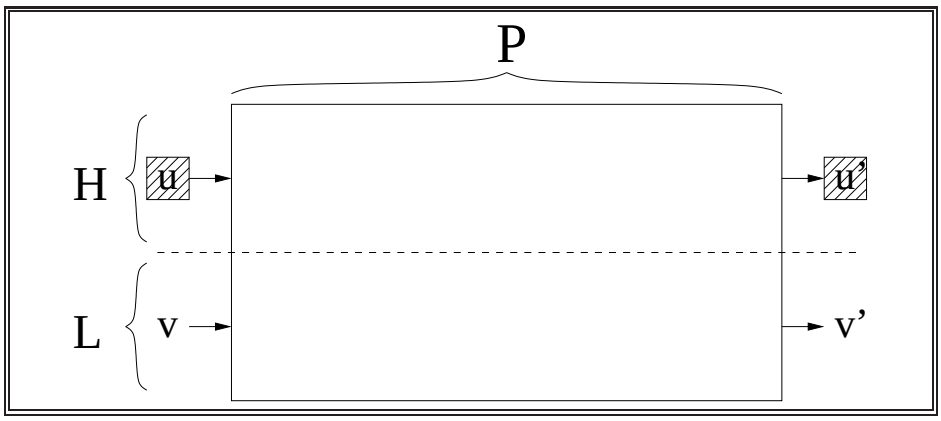


Figure 2.3: The information flow model.

immaterial).

$$\Gamma \vdash 3 : \texttt{int}$$

An inference rule composes the results of the typing process carried out on pieces of a program, i.e., the rule premises, to type the program itself, i.e., the rule conclusion. High-order types, e.g., those denoting functions and data structures, are obtained in this way. For instance, the following rule types a constant function.

$$\frac{\Gamma \cup \{x \mapsto \texttt{int}\} \vdash 0 : \texttt{int}}{\Gamma \vdash \texttt{fun}(\texttt{int}\ x) = 0 : \texttt{int} \to \texttt{int}}$$

The arrow type $\texttt{int} \to \texttt{int}$ denotes a transformation from the (explicitly typed) input x to the constant 0.

Several authors proposed security frameworks using type systems and their extensions. David Walker [182] proposed a *certifying compiler* for the verification and enforcement of safety properties. The compiler of [182] generalises the concept of type safety by allowing the developers to type check their programs against arbitrary policies. Also, the compilation procedure includes an instrumentation step which is responsible for adding security dynamic checks to the program code.

Volpano et al. [181] introduce a type system for proving properties of programs written in a sequential imperative language. Then, in [174] they also extend their approach to a concurrent computational model. Their approach mainly deals with *information flow* [32, 76, 77]. Briefly, information flow takes place when an observer which should only know a *low* (L) level information can infer the *high* (H) level data handle by a program. Figure 2.3 schematically depicts this scenario. The type system of [181] can deal with *explicit* flow as well as *implicit* flow [1]. Roughly, an information flow is said to be explicit when it consists of the direct migration of data stored in high variables to low variables. For example, an explicit flow is caused by the assignment of (expressions containing) high variables to low variables. Instead,

implicit flow happens when an observable difference in the low output is caused by some high values.

Type and effect systems [140] represent a refinement of the classical type systems. They enrich types with special annotations. The annotations represent side effects that programs generates during their execution. These annotations can be suitably used for representing the execution traces of programs necessary for implementing history-based security mechanisms.

For instance, Skalka et al. [172, 173] presented a type and effect system for the extraction of (over-approximations of) the execution histories of programs. Also, their type system is used to evaluate security predicates defined over the histories in order to statically check whether they are satisfied by the current implementation.

A similar approach has been proposed by Bartoletti et al. [24, 25] for the extraction of *history expressions* from the programs source code. Briefly, history expressions denote sets of execution histories instrumented with *policy framing* operators. A policy framing defines the local scope of a policy over a part of the execution. When a program enters a policy scope, its history is valid if and only if the policy is respected until the closure of the framing. In their model, security policies are expressed through *usage automata* (see Section 2.3). History expressions annotated with policy framing can be statically evaluated for checking their validity before the execution and used to produce an optimised runtime monitor that only checks the policies which have not passed the verification step.

2.2.2 Model checking

The *model checking* [57] was originally proposed in the '80s [56] as a fully automatic approach to program verification. The main statement of the model checking problem is a quite simple question: *is it true that a certain model M satisfies a specification φ?* In symbols

$$M \models \varphi$$

The model M represents an actual system, e.g., a program. Traditionally, M is defined through a finite-state transition system, e.g., a *Kripke structure*. Roughly, a Kripke structure consists of a set of edges and a set of labelled states. A label is a list of atomic propositions that are true in a certain state.

Instead, the specification φ expresses a temporal properties that must be verified over M. Typically, φ is written in some temporal logic, e.g., LTL or CTL* (see Section 2.3).

In practice, the verification of φ is an exhaustive visit of the states of M starting from an initial state. Each step evaluates φ against the atomic propositions of the current state s and iterates once for each state that is reachable from s. The iteration consists in solving the model checking problem starting from the new state and evaluating a new property obtained after unfolding the temporal operators of φ according to the executed transition. We write $M, s \models \varphi$ to emphasise that the

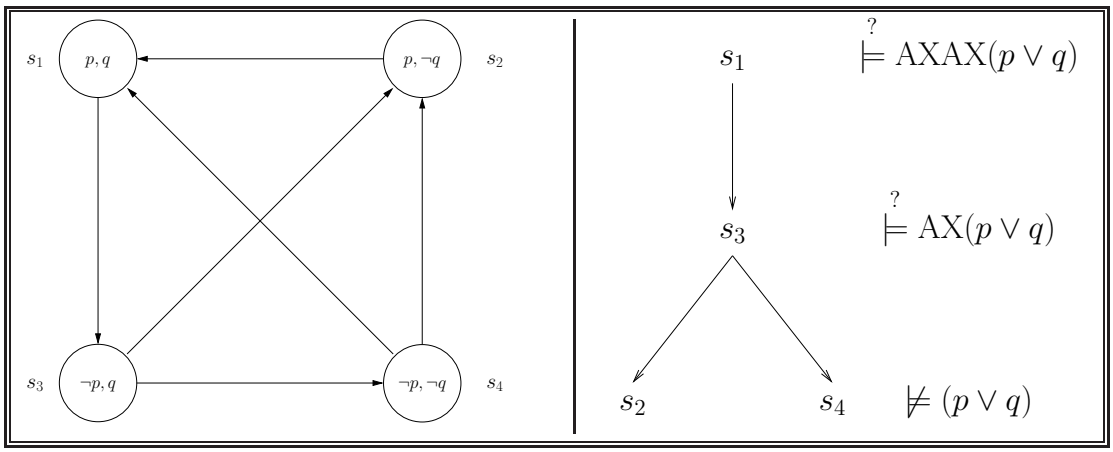


Figure 2.4: A Kripke structure and the visit of a model checker.

model checking algorithm is evaluating φ on the state s of M.

In general, the procedure is inductively defined by declaring the semantics of the connectives of the used temporal logic. For instance, consider the CTL AXφ operator meaning that φ must hold in all the states reachable from the current one with a single step. The corresponding rule is:

$$M, s \models \text{AX}\varphi \quad \Longleftrightarrow \quad \text{for all states } s' \text{ such that } s \rightarrow s' \ : \ M, s' \models \varphi$$

An example can better clarify the behaviour of the model checking approach.

Example 2.1 Consider the Kripke structure M depicted in Figure 2.4 (left side). It has four states, i.e., s_1, s_2, s_3 and s_4, that are labelled with the atomic propositions p and q (we used their negation to label the states where they do not hold). Now, we want to verify whether the formula AXAX$(p \vee q)$ is valid on M starting from the state s_1, i.e., we ask whether $M, s_1 \models \text{AXAX}(p \vee q)$. According to the rule introduced above, the formula is satisfied if and only if the sub-formula AX$(p \vee q)$ is valid in the states that are reachable with a single move from s_1, i.e., only s_3. We apply again the rule to s_3 and AX$(p \vee q)$. Both s_2 and s_4 are pointed by a transition rooted in s_3. However, we can easily check that $s_2 \models (p \vee q)$ (as p holds in s_2), while $s_4 \not\models (p \vee q)$. The right side of Figure 2.4 schematically represents the procedure described above.

The previous example outlines an important aspect of the model checking approach. The output of a model checker can be either true, i.e., the model complies with the given specification, or a *counterexample*. A counterexample is a path in the model that violates the specification. Needless to say, counterexamples are very useful for correcting design errors.

The main limitation to the practical application is that the model checking problem is computationally hard [169]. For instance, solving the model checking problem

for a LTL specification is exponential in size of the formula (and also linear in the size of the model). Hence, many authors worked on making the model checking problem tractable. Some of the major proposals are *symbolic model checking* [47] and *bounded model checking* [37]. Briefly, the first exploits an abstract representation of groups of states for reducing the search space, while the second bounds the length of the visited path to a maximum value.

Model checking is not the only approach being proposed for the formal verification of the program properties. Other techniques can be applied for the same purposes. Among them, we list some of the most important.

- *Automatic theorem proving.* It consists of a logical framework that, starting from a given logical formula, verifies its validity. Theorem provers have been successfully applied to software and hardware verification (e.g., see [153, 15]).

- *Abstract interpretation.* It is based on a sound approximation of the computation of a program. In several works, e.g., [135, 139], it has be used for program verification and in particular for control and data flow analysis.

- *Protocol verification.* It automatically verifies whether a security protocol can be violated. For instance, the resistance of the protocol can be verified by modelling an intruder with various capabilities [126, 127].

2.3 Trace properties specification

As we said above, in history-based security the formalisms used for the specification of histories properties play a central role. In this section we present some of the most used specification formalisms for defining properties of execution traces.

2.3.1 Automata-based Specification

Many authors propose variants of *finite state automata* [5] for the specification of the properties of the execution histories. A finite state automaton is a reader accepting the sequences of symbols belonging to a certain language. Intuitively the sequences accepted (or rejected) by an automaton are the legal ones.

The languages accepted by finite state automata are called *regular* as they coincide with the languages of regular expressions. Regular languages have several properties of interest, e.g., they are closed under union, intersection and complementation. These properties often correspond to desirable properties for security properties, e.g., conjunction, disjunction and negation, which finite state automata enjoy.

One of the major automata-theoretic proposals appeared in [8] and was then extended in [168]. In particular, Schneider's security automata [168] have been presented as a formalism for defining security monitors. Briefly, the security monitor

simulates the transitions of the Schneider's automaton by stimulating it with the observed execution trace. If the automaton transits to a final state, the monitor halts its target.

Interestingly, the class of properties that security automata can express is that of *safety properties*. Roughly, safety properties are those saying that the computation never reaches a dangerous state. This class of properties is of primary importance, e.g., for the dynamic monitoring of the usage control policies.

Then, Ligatti et al. [30, 123] studied the possibility of enforcing a wider class of security policies, namely *edit properties*, through special automata, namely *edit automata*. Edit automata are defined through special functions that evaluates the execution of a target action by action. For each action these functions can decide to allow it, to interrupt the execution (*truncation*), to void the action (*suppression*) or to add an extra one (*insertion*). We will provide the reader with further details about edit automata in Section 4.2.

Bartoletti et al. [25, 24] advocated *usage automata* as a formalism for defining security policies having a *local scope* over the histories of programs. Roughly, the definition of usage automaton do not differ very much from that of Schneider's automaton. They both have a set of states, an input alphabet and a set of labelled transitions. However, usage automata have a number of features that make them suitable for many aspects of resources usage analysis. In particular, their transitions are labelled with *parametric actions* rather then plain symbols. In this way, they can model in an intuitive way the invocation of functions and methods over security-relevant resources.

Also, resources can be represented using variables and their complement, e.g., x and $\bar{x}$ respectively. Variables are useful for having equality and difference checks on the resources appearing in the automaton transitions. Indeed, two actions referring to the same variables must be instantiated on the same resources. Instead, the complement means that the action is evaluated for any resource, but the one that is complemented.

Being parametric over actual resources, each usage automaton represents a class of possible security automata, i.e., one for each possible instantiation of its resources. Evaluating a usage automaton against an execution trace amounts to say that the automaton is instantiated over all the possible bindings between its variables and the actual resource appearing in the trace. Then, the policy is verified if none of the instantiations reaches a final, offending state. Moreover, in [27] they have been extended to deal with the dynamic creation of resources. This aspect is quite important especially for the modern programming languages that often exploit the instantiations of classes, i.e., *objects*, of resources.

Another useful property is that usage automata only contain transitions for the actions of interest. All the other actions cause an automaton to loop on its current state. In this way, a policy designer can focus only on the actions that are considered to be critical. We propose the following example to better explain the structure and

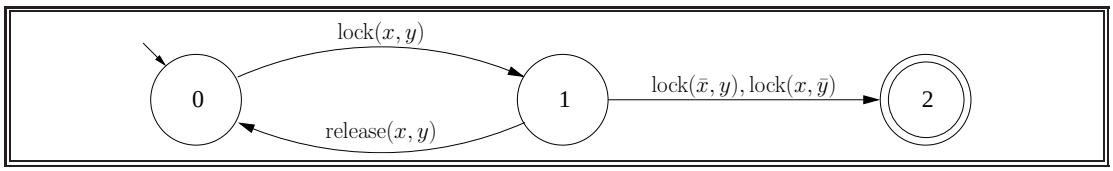


Figure 2.5: A usage automaton for a multi-task environment.

behaviour of usage automata.

Example 2.2 Imagine a multi-task system where a finite set of resources must be shared among the running processes. Each process is an agent performing some computation possibly involving the access to one or more resources. In order to avoid conflicts a process must *lock* a resource before its usage and then *release* it. We want to apply the following policy: "a process can only access one resource at a time and the access to a resource is mutually exclusive". Assuming to have, among the others, two actions $\text{lock}(P, r)$ and $\text{release}(P, r)$ meaning that a process P locks or releases a resource r, the resulting automaton is shown in Figure 2.5.

Consider now an environment with two processes, i.e., P and Q, and two resources, i.e., r and r'. In this case the usage automaton denotes all the security automata arising from the instantiations of its parameters. Excluding the inconsistent ones, e.g., $x \mapsto P$ and $y \mapsto Q$, the automaton instantiations are:

1. $x \mapsto P$, $y \mapsto r$;
2. $x \mapsto Q$, $y \mapsto r$;
3. $x \mapsto P$, $y \mapsto r'$;
4. $x \mapsto Q$, $y \mapsto r'$.

If P locks r the first automaton moves to state 1, while the others loop on state 0. Then, if Q tries to lock r, the second automaton moves to state 1, the third and the fourth loop on 0, and the first one reaches the final state 2 raising a security error.

Interface automata [71] have been presented as a formalisms for describing behavioural interfaces. Originally, they were proposed for defining the temporal interface of a software component in terms of the methods that it exposes and invokes.

Basically, an interface automaton is defined through (i) a set of states, (ii) an alphabet of symbols and (iii) a set of labelled transitions. The alphabet is the union of three, disjoint sets of *input*, *output* and *internal* symbols. Input symbols represent interface methods that can be invoked from outside, output symbols denote the methods that the automaton can invoke and internal symbols are the private methods. Transitions can be labelled with any element of the alphabet. The following example shows a behavioural interface defined through an interface automaton.

Example 2.3 Consider the interface automaton of Figure 2.6. It represents the interface of a manager for TCP communications using UDP primitives. Briefly, its method interface is denoted through the external frame. Incoming arrows are the module methods while outgoing ones are the methods it invokes. Interface actions labelling the transitions are annotate with ? if they refer to own methods and ! if they are external invocations. Internal operations, i.e., actions that cannot be triggered by or to the automaton, are annotated with ;.

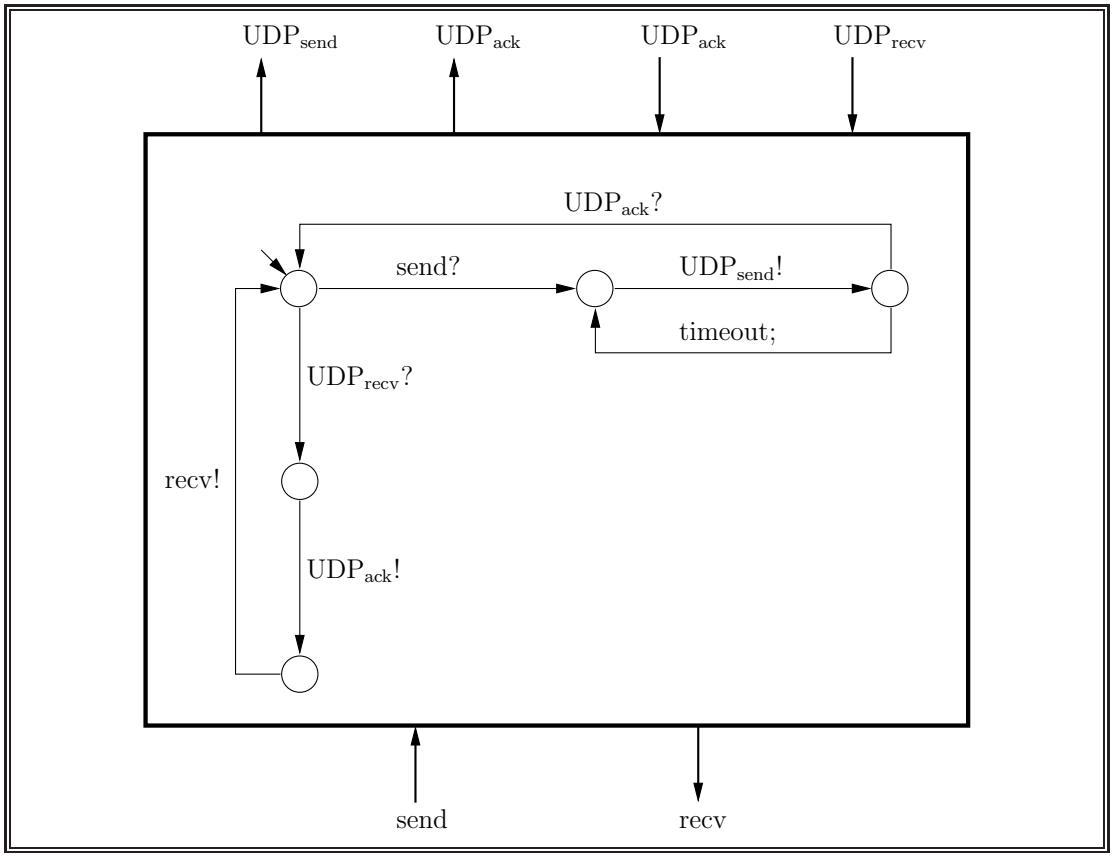


Figure 2.6: An interface automaton for the TCP protocol.

Starting from the initial state, the manager can either send a message (horizontal path) or receive a UDP packet (vertical path). In the first case, it sends the message using the sending facility provided by the UDP protocol and waits for a UDP acknowledgement before returning to the initial state. If an internal timeout is produced before the expected acknowledgement, the automaton repeats the sending action. Instead, if a UDP packet is received, the automaton answers with an acknowledgement and invokes the TCP receive method.

The trace semantics of interface automata is given it terms of *execution fragments*. Basically, an execution fragment is an alternating sequence $s_0\alpha_1 s_1 \cdots \alpha_n s_n$

of states and symbols representing a computation that starts from state s_0 and terminates in state s_n. During the computation, the automaton touches the states and executes the methods appearing in the fragment.

Even though they were not proposed for defining execution properties, interface automata can be exploited for this purpose. Also, interface automata have a number of properties, especially related to compositional aspects. For instance, they can be sequentially composed through their input/output interfaces. In Section 4.2.2 we propose an application of interface automata to the security issues.

2.3.2 Process Algebras

In the last decade, the impact of *process algebras* for the formal specification have been constantly growing in the security community. Similarly to finite state automata, process algebras offer an operational representation of the expected behaviour of a program. Usually, the agents of a process algebra are defined through a simple syntax that is able to focus on some relevant aspects of the computation. For instance, consider the syntax of the *calculus of communicating systems* (CCS) [136] given below.

$$P, Q ::= \mathbf{0} \mid \alpha.P \mid P + Q \mid P \mid Q \mid P[f] \mid P \setminus L$$

A process can be the inactive one ($\mathbf{0}$), an agent with an action prefix ($\alpha.P$), a nondeterministic choice between two processes ($P + Q$), a parallel composition of two agents ($P \mid Q$), an relabelling of an agent ($P[f]$ where f is a substitution function over the actions of P) or a restriction ($P \setminus L$ where L is a set of actions). Clearly the CCS mainly focuses on certain aspects of the computation. In particular, it can be easily applied for defining concurrent agents, communicating and synchronising through special actions. As a matter of fact, prefixes can be output as well as input actions, e.g., a and $\bar{a}$, or even the silent action τ.

The semantics of a process algebra is often defined in terms of a *labelled transition system* (LTS). An LTS has a set of states representing possible configurations of a process. The computation causes state changes according to a set transitions, each of them being labelled with an action. Inference rules are used to define the behaviour of the transitions. For instance, consider the following rule for parallel composition of CCS processes.

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

It states that two concurrent processes can synchronise on a proper pair of input and output actions to perform a silent transition. Needless to say, LTSs fit with the history-based security model and have been largely exploited for this purpose.

The π-calculus [137] was proposed as an extension to the CCS for dealing with processes that can exchange messages rather that simply synchronising. Interestingly, it turned out that the π-calculus is a universal model of computation, i.e., it

is Turing-complete. Hence, it offers the richest expressive power for the operational specification of processes.

An interesting extension of the π-calculus has been proposed by Abadi and Gordon [3]. Their spi-calculus extends the π-calculus with facilities useful for modelling cryptographic protocols. For instance, they can model encryption keys generation and exchange. In this way they can apply formal verification techniques for checking the protocol resistance, e.g., against private key disclosure.

A very peculiar feature of process algebras is the number of *simulation relations* which have been introduced in the literature. Roughly, a simulation is the relation occurring between two agents such that the first can reproduce the behaviour of the second. The behaviour is considered in terms of actions produced by the computation. For instance, the simulation relations can differ for their assumptions on the observable actions. When two processes simulate each other it is called a *bisimulation* and the two agents are said *bisimilar*. Saying that a process simulates another one has important consequences, e.g., in terms of components substitutability. This reasoning is used, for example, in *action refinement* [162] for moving from an abstract model of process toward its implementation.

In [130] the authors combine simulation and *partial model checking* [10] to automatically synthesise program controllers. More in detail, starting from a system specification and a security policy they find a controller that guarantees the policy compliance on the interaction with any possible malicious component. Their controllers are *maximal* in the sense that for each other controller enforcing the same security policy theirs is more general (with respect to a simulation relation).

Process algebras have been also used as the basis for defining languages for the specification of security properties. For example, the *policy specification language* (PSLang) [86] has been presented for specifying security policies for reference monitors. PSLang uses a Java-like syntax for defining an event-driven computational model. In the style of process algebras, a PSLang policy declares a set of events on which it can synchronise its computation. When one of these events is received, the PSLang process updates its security state, possibly leading to raising a security error.

2.3.3 Modal and temporal logics

Modal logics extend standard logics, e.g., propositional, first-order logic, with *modality* operators. Modalities apply a special context to a logical formula. For instance, we can have modalities dealing with possibility and necessity, rights and duties, knowledge and belief. Modalities proved to be particularly good at modelling several properties of computing systems.

For instance, the *Hennessy-Milner logic* [98] uses two special modalities, i.e., universal and existential quantifiers, for reasoning about the transitions that a process, e.g., a CCS agent, can perform. The two modalities are also annotated with actions that restrict the evaluation of a property on the transition with the right label only.

The modal operators are $[a]$, namely "box", and $\langle a \rangle$, namely "diamond". Both quantify a certain formula. The former states that each computation that starts with an a transition must respect the quantified formula, while the latter requires that at least one such computation exists.

Another major proposal is the modal μ-calculus [113]. Basically, the syntax of the μ-calculus comprises the two modalities described above, i.e., box and diamond, and two operators for least and greatest fixed point, i.e., μ and ν respectively. However, the extreme expressiveness of the μ-calculus impacts on its usability. Indeed, the satisfiability problem for the μ-calculus is EXPTIME-complete [183]

Probably *temporal logics* [128] are the most used technique for the specification of the temporal properties of programs and, often, also for usage control [188]. Temporal logics are a category of modal logics where modalities refer to the temporal context of a formula. Among them, *linear temporal logic* (LTL) [157] is one of the most used. Basically, LTL uses the temporal operators `X`, `F`, `G` and `U` for defining temporal properties. Respectively, they mean that a formula φ must hold in the next state ($\mathtt{X}\varphi$), eventually in the future ($\mathtt{F}\varphi$), globally during the computation ($\mathtt{G}\varphi$) or until a second formula ψ is verified ($\varphi\mathtt{U}\psi$).

Also, the LTL syntax has been extended with the operators of the *computation tree logic* (CTL) [55] for dealing with the properties of branching computations. The logics obtained in this way is called CTL*.

2.4 Composed systems security in the field

The continuous evolution of the telecommunication technology is a common factor of many recent changes of the computing environments. For instance, in conjunction with the advancements in the miniaturization of the hardware components, it made possible the revolution of mobile devices that we are currently experiencing. These conditions influenced the current trend for mobile applications. Usually, mobile applications are low-cost (or even free) pieces of software which have been designed and developed for being easily distributed and deployed on possibly resource-limited platforms. This very successful paradigm caused the creation of a number of digital marketplaces dedicated to the commercialization of the mobile applications.

On the other hand, new distributed computational paradigms deeply influenced the design of the web-based systems. In this field, terms like *Service Oriented Computing*, *Grid Computing* and *Cloud Computing* are becoming more and more common. Roughly, they allow complex services and tasks to arise from the composition and the efforts of many, distributed components. The sub-components can be the result of other, arbitrary complex compositions.

These two worlds, i.e., mobile applications and distributed systems, also integrate each other. Indeed, mobile devices represent a continuous access point to the world of services and remote business applications that, on the other way round, provide the users of the devices with their advanced computational capability.

This complex scenario offers several opportunities of security attacks. As it involves so many different aspects and conditions, providing the proper protection mechanisms is both crucial and extremely hard. Here we briefly survey on the state-of-the-art technologies and models for dealing with the security of these applications.

2.4.1 J2ME security

Java [16, 124] is the leading technology for the portability of software. Using Java, the software developers can compile their applications into an intermediate language, namely *bytecode*, which can be executed on every platform running the proper interpreter, i.e., the *Java virtual machine* (JVM). The success of the Java framework for mobile devices, i.e., the *Java micro edition* (J2ME), is mainly due to this feature allowing the execution of the same application on many, totally heterogeneous platforms.

Optimized versions of the JVM, e.g., the KVM [177, 105, 178], are present on most of the existing devices. However, the increased capabilities of the new generation devices make it possible to use some almost fully featured versions of the Java runtime.

Java exploit several static mechanisms for verifying the correctness of programs. Basically, Java is a strongly typed language. Typing a Java program, its developer is provided with a proof that several errors, e.g., stack overflow and cast exception, will not happen at execution time. Moreover, some proposals exist for the static verification of other security properties, e.g., JIF [103] for information flow properties (see Section 2.2.1). In this scope, JML [118, 48] received major attention. Roughly, JML allows one to specify properties that must be respected by the methods of a class directly within the comment lines. At compile-time the properties are automatically checked against the real implementation.

The effectiveness of these techniques is seriously compromised under the mobility assumptions that we are considering. Indeed, here we assume that (i) the user receives the application through (from) some unreliable channel (repository) and (ii) rarely users and providers have different security requirements. As a consequence, often the guarantees that the developers obtain from the static techniques are not extended to the software user.

Nevertheless, dynamic techniques can be suitably applied to the Java programs after their deployment. Among them, *stack inspection* [86, 91] is probably the most used. The stack inspection model works as follows. When the execution enters the scope of a method, i.e., after an invocation, a new entry for it is pushed on the method stack. Each method has a set of access privileges. When the program tries to access a resource requiring certain privileges, the access is granted if and only if the current stack is compatible. In other words, the permission depends on whether every method in the stack has the needed privileges.

Other solutions aim at extending the validity of the static checks performed at development time to the platform of the code consumer. In particular, the idea

is to mark the code with annotations describing the kind of checks performed at compilation time. This process is carried out by a *preverifier* [178]. The preverifier checks some properties of the compiled application, e.g., control-flow safety, and annotates the class files with additional information. These annotations are used by the client's virtual machine for implementing a fast check on the properties that have been verified by the code provider.

The integrity of the code (possibly with annotations) is guaranteed through package certification. Briefly, the software releasers acquire a certificate from one of the *Certification Authorities* (CAs) accepted by the (manufacturer of the) platform their software must run on. Then, they use the certificate information to sign their software. At deploy-time, the platform checks the signature integrity, i.e., whether the content of the package changed after its compilation, and assigns the application to a security domain. Often, the security domain only depends on the identity of the CA releasing the certificate.

Even though J2ME is becoming an obsolete technology for mobile applications, the number of deployment paradigms using mobile applications is growing, e.g., see [94, 54, 13]. Many of these systems use Java or Java-like development frameworks. From the security perspective, they often rely on traditional approaches, e.g., code signature and stack inspection [93]. Hence, the reasoning on the security of J2ME can, in most cases, be applied to other, similar contexts.

2.4.2 Web services and grid computing

Nowadays, Service-Oriented Computing (SOC) [150, 148, 149, 147] is a consolidated paradigm for designing and implementing indefinitely complex systems is a constructive, compositional and distributed manner. The success of the web services drastically changed the way we understand the web-based applications and software in general. Such a rapid change in the web-based software development paradigms can offer new opportunities as well as new threats. Naively, one could think that the main risk factor for the web services is the unreliable network, i.e., the Internet, that they use for communications and compositions. However, it has been pointed out [147] that the main concerns about security are at application level.

Several new standards have been introduced for dealing with the different security issues involved in service composition. The foundational standard that has been defined by the web services community is *WS-Security* [40]. Basically, WS-Security extends the Simple Object Access Protocol (SOAP) with security-related technologies, e.g., cryptographic primitives. Mainly, WS-Security provides the service designer with facilities for dealing with the integrity and confidentiality of communications and sessions. Other standards are built upon WS-Security. Among them, we cite *WS-Policy* and *WS-Trust*. Briefly, WS-Policy defines an XML-based language for declaring security requirements over the service sessions. For instance, a WS-Policy specification can require messages to be encrypted with a certain technique or key size. Instead, WS-Trust is used for managing trust relations which can

subsist among services.

Business processes offer a representation of service compositions that can be exploited for security purposes. For instance, *control flow* and *data flow* analysis can be performed once a business process has been defined. Roughly, control flow describes how the computation is carried out, which platforms execute it and in which order. Similarly, data flow gives a representation of the way information and data structures move within the service composition. The standard way for modelling web services business processes is the *business process execution language* (WS-BPEL or BPEL4WS) [12, 107].

Even though a significant efforts have been done to cope with the security of web services, the definition and development of their security specifications is still a work in progress. Consequently, also the formal security techniques for web services suffered from this absence of suitable specification methods.

Grid computing [90] (and also its descendant, cloud computing) is one of the main technologies for highly distributed computing systems. Roughly, it aims at holistically organising a group of nodes, namely a virtual organisation, sharing their resources for achieving a goal which is too hard or costly for a single node. The security issues affecting the computational grid, e.g., see [89], are somehow similar to those seen for the web services. However, as they rely on resource-sharing, some authors, e.g., see [106, 112], outlined that introducing usage control mechanisms for the grid is a priority task and presented their proposals. To the best of our knowledge, the grid community did not propose any standard for the application-level security issues. Indeed, most of the work about security has been dedicated to the problem of establishing secure communications at network level.

Mobile Application Security and Enforcement on Devices

In this chapter we present our work and results about the security of *mobile applications*. A mobile application is a program which is developed in an environment and deployed in a different one. The actual deployment happens only after a transmission step during which the *code producer* sends the application to the *code consumer*. From the security perspective, several threats can affect this process. Here we focus on a very precise aspect: *which precautions can help the code producer (consumer) to protect the resources of his application (platform)?*

Clearly, the possible attacks and countermeasures change with the chosen observer, i.e., producer and consumer. For instance, a code consumer could be interested in preventing the applications from stealing private data, while the producer could require protection against the presence of other, malicious software running on a compromised platform.

For dealing with the security issues, we often need to start from some assumptions. Most of them concerns the *trusted computing base* (TCB) [115], i.e., the (small) set of software and hardware components which we assume to work as expected. Hence, these elements are assumed to never fail or misbehave. We briefly present the assumptions on the TCB that we used in this chapter.

We start in Section 3.1 by considering the point of view of the code producer. In particular, we present an extension of the Java programming language [93] for annotating the source code with security policies. In this case, the software developer assumes that the program interpreter, i.e., the Java Virtual Machine, is part of the TCB. Instead, other components, e.g., programs and libraries, may perform some security violations. Then, in Section 3.2 we present an implementation of this model based on a secure compiler including a verification process and the instrumentation step, called *inlining*.

In Section 3.3 we move on the other side, i.e., we study a mechanism for protecting the code consumer. We assume a realistic applicative context, i.e., mobile devices, where an application can be imported in a platform and executed. The code consumer can deploy a security controller, belonging to the TCB, for executing the (untrusted) applications in a safe way. Again, we exploit an inlining-based approach to instrument the applications with security controls. Finally, in Section 3.4 we ex-

tend this model in order to create a security environment involving all the activities taking place on mobile devices.

3.1 Extending Java with local policies

Two main aspects of security are particularly relevant and sensitive during the application development process: (i) the design of security policies and (ii) their application. Designing a security policy may require a precise knowledge of many technical issues. Moreover, once it has been defined, applying a security policy is still a sensitive operation. Nevertheless, both these issues can be simplified through a modular reasoning.

Object-oriented programming represents a perfect context for dealing with the security of a program in a compositional way. As a matter of fact, *classes* are software compounds consisting of operations, i.e. *methods*, sharing some data, i.e. *fields*. In general, classes are designed in order to contain all and only the information regarding a precise aspect of the computation. Also, reasoning about the security of a single class is much easier than doing it for a whole program, i.e. a group of many classes.

Some authors proposed to integrate the source code of the Object-oriented languages with special annotations (also) for security requirements. For instance, [118] introduces the *Java Modelling Language* (JML) for defining method requirements within the code comments. These special annotations can be preprocessed for verifying whether they are actually satisfied by the program implementation they refer to [88].

A similar approach, namely *Chalice*, has been proposed by [119]. Chalice can deal with the requirements of objects and methods. Moreover, it can be suitably applied for verifying that concurrently executing objects, i.e. threads, do not violate the specifications.

These proposals outline the growing interest for providing the software designer and developers with automatic mechanisms for verifying the correctness of their applications. Such mechanisms must support the software creation process without forcing the developers and designer to acquire many new technical skills.

In the following we present the design of an extension to the Java language, so to enhance its security mechanism with local policies [21]. Local policies are orthogonal to Java code and their specification is produced, together with the application code, during the development process. Policies are defined through usage automata, i.e., a special category of security automata, where the input alphabet of security-relevant events coincides with the set of the controlled Java methods. These methods can be Java APIs as well as members of the classes composing the application.

3.1.1 Local policies specification

We are interested in specifying and enforcing safety policies of *method traces*, i.e. the sequences of run-time method calls. We define policies through usage automata featuring facilities to deal with method parameters in order to adapt them to the Java framework. The first step consists in defining a mapping between observable method invocations and the input alphabet of a usage automaton. Then we must define a proper syntax for declaring the structure of a usage automaton directly inside the Java source code. Finally, we instantiate these definitions into actual usage automata.

A motivating example. Consider a trusted component `NaiveBackup` that offers static methods for backing up and recovering files. Assume that the file resource can be accessed through the following interface:

```
public File(String name, String dir);
public String read();
public void write(String text);
public String getName();
public String getDir();
```

The constructor takes as parameters the name of the file and the directory where it is located. A new file is created when no file with the given name exists in the given directory. The meaning of the other methods is as expected. In the class `NaiveBackup`, the method `backup(src)` copies the file `src` into a file with the same name, located in the directory `/bkp`. The method `recover(dst)` copies the backed up data to the file `dst`. As a naïve attempt to optimise the access to backup files, the last backed up file is kept open.

```
class NaiveBackup {
  static File last;
  public static backup(File src) {
    if(src.getName() != last.getName())
      last = new File(src.getName(), "/bkp");
    last.write(src.read());
  }
  public static recover(File dst) {
    if(dst.getName() != last.getName())
      last = new File(dst.getName(), "/bkp");
    dst.write(last.read());
  }
}
```

Consider now a malicious `Plugin` class, trying to spoof `NaiveBackup` so to obtain a copy of a secret passwords file. The method `m()` of Plugin first creates a file called `passwd` in the directory `/tmp`, and then uses `NaiveBackup` to recover the content of the backed up password file (i.e., `/bkp/passwd`).

```
class Plugin {
  public void m() {
    File g = new File("passwd","/tmp");
    NaiveBackup.recover(g);
  }
}
```

Clearly, this attack only takes place if the last argument of the `backup` method was a file called `passwd`. However, it is important to note that this kind of behaviour cannot be detected though stack inspection [93]. Indeed, when the method `recover` is invoked, the stack contains no references to `backup` and vice versa. Instead, as our approach is history-based, we can identify this security violation.

Aliases and events. As mentioned above, our policies constrain the sequence of run-time method calls. Clearly, any policy language needs some facilities to abstract from the (possibly infinite) actual parameters occurring in the programs executions. To do that, one could use method signatures as a basic building block to specify policies. However, this may lead to unnecessarily verbose specifications; so, we provided our policy language with a further indirection level, which helps in keeping simple the writing of policies.

We call *aliases* our abstractions of the security-relevant methods. Let $\mathtt{m}$ be a method of class $\mathtt{C}$, a signature for such a method has the form

$$(\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{C}_1\ \mathtt{y}_1, \ldots, \mathtt{C}_n\ \mathtt{y}_n)\ :\ \mathtt{C}'$$

where $\mathtt{y}$ is the target object, $\mathtt{y}_i$ are the parameters (having type $\mathtt{C}_i$, for $i \in \{1 \ldots n\}$) and $\mathtt{C}'$ is the return type.

An alias $\mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ for $\mathtt{m}$ is defined as:

$$\mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k) := (\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{C}_1\ \mathtt{y}_1, \ldots, \mathtt{C}_n\ \mathtt{y}_n)$$

where $\mathtt{ev}$ is an identifier and $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\} \subseteq \{\mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_n\}$. This set inclusion may be strict, when some parameters in the method signature are irrelevant for the policy of interest. Note that here we ignore the returning value of methods. However, this model can be easily extended to also consider them. A solution could be using a second alias with only one parameter for gaining the visibility of the results of the method invocations.

Example 3.4 Consider the following aliases:

```
read(f)  := (f:File).read()
write(f) := (f:File).write(String t)
new(f,d) := (f:File).<init>(String n, String d)
```

The first item means that `read(f)` is an alias for the method `read` of the class `File`. The only parameter of the alias is the target object of the method, that is the object `f` of type `java.io.File` (we avoid to use fully qualified class names when there is no risk of ambiguity). Similarly, the alias `write` corresponds the the invocation of the method `write`. Just note that the alias does not refer to the parameter `t` that instead is neglected. Finally, the last alias associates the name `new` to the method `<init>`, i.e., the default constructor, of the class `File`. Again, the alias ignore one parameter (namely `d`) of the method.

Note also that aliasing is not an injective function from method signatures to aliases. For instance, if multiple methods were provided to write a file, they could be abstracted to a single alias `write(f)`, so simplifying the definition of the usage automaton.

As a method call is the concrete counterpart of a method signature (the formal parameters in the latter are concretized into actual parameters in the former), an *event* is the concrete counterpart of an alias. Summing up, we have a mapping from method calls to events, and – by lifting this to sequences – a mapping from method traces to event traces. For instance, applying the aliases of Example 3.4, the method trace:

```
g.<init>("passwd","/etc") f.read() g.write("secret")
```

is abstracted into the event trace:

```
new(g,"/etc") read(f) write(g)
```

Usage automata definition. We extend the standard structure of usage automata by adding *guards* to the automata transitions. As usual, variables represent universally quantified resources. Instead, guards express conditions among resources. Hence, a usage automaton is specified in plain text as follows:

`name` : *name of the usage automaton*
`aliases`: *list of aliases* A
`states` : *set* Q *of states*
`start` : *initial state* $\imath \in Q$
`final` : *final (offending) states* $F \subseteq Q \setminus \{\imath\}$
`trans` : *set of labelled transitions* $T \subseteq Q \times E \times G \times Q$

The first item is just a string, to be used when referencing the usage automaton (e.g. in the definition of sandboxes, see below). The second item lists the aliases (separated by newlines) relevant for the policy. The remaining four items define the operative part of the usage automaton, i.e. its finite set of states Q (separated by spaces), the initial state $\imath$, the set of final states F, and the set of edges or transitions T. The formal parameters of a usage automaton are the variables occurring in its edges.

To define an edge from state `q` to state `q'` we use the following syntax:

`q --` *label* `--> q' when` *guard*

The *label* is a (possibly, partial) concretization of an alias defined in the `aliases` item. Concretizing an alias $\texttt{ev}(\texttt{x}_1, \ldots, \texttt{x}_k)$ results in a label of the form $\texttt{ev}(\texttt{Z}_1, \ldots, \texttt{Z}_k)$, where each $\texttt{Z}_i$ can be either:

- `S`, for a static object `S`. Static objects comprise strings (e.g. `"/tmp"`), final static fields (e.g. `User.guest`), and values of enum types.
- `x`, for some variable `x`. This means that the parameter at position `i` is universally quantified over any object.
- $*$, a wildcard that stands for any object.

A *guard* represents a condition among resources, defined by the following syntax:

guard `::= true | Y != Z |` *guard* `and` *guard*

where `Y` and `Z` are either static objects or variables. The guard `when true` can be omitted. For the sake of minimality, we introduced only two logical operators: inequality between resources and conjunction. Several other operators do not need to be explicitly declared as they are implicitly defined in the syntax of policies. For instance, we only use inequality (i.e. `!=`) as equality is implicitly defined by the syntax of labels and we can abbreviate `X != X` (for some `X`) with `false`. Also, we do not need to introduce an `or` operator as we can simulate its behaviour by replicating a transition and guarding its instances with the guards under disjunction.

Note that policies can only control methods known at static time. Indeed, the definition of a policy must be done before the first execution of its target. In the case of dynamically loaded code, where methods are only discovered at run-time, it is still possible to specify and enforce interesting classes of policies. For instance, system resources – which are accessed through the JVM libraries only – can always be protected by policies. We can also refer to the methods of interfaces and abstract classes. In this way, whenever a new class implementing a known abstract method is loaded at run-time we can control its invocations.

Thus, the policies must be attached to an application source code when the compilation process starts. A suitable solution consists in extending the syntax of

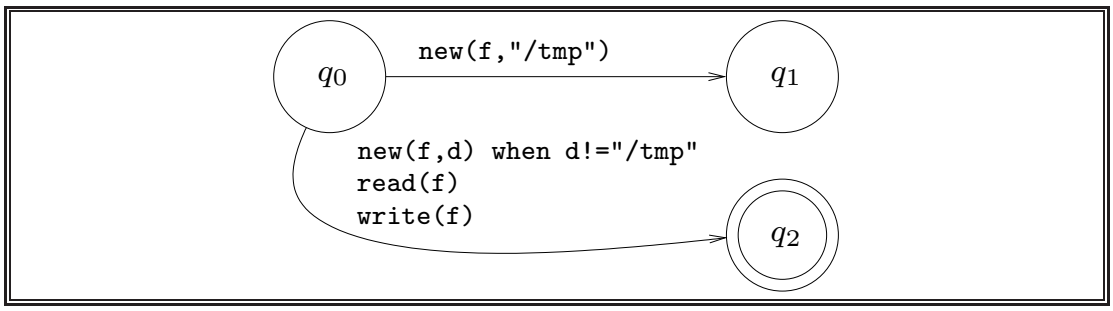


Figure 3.7: File confinement policy `file-confine(f,d)`.

the *Java Modelling Language* (JML [118, 48]). JML is an extension of the standard Java comments offering a rich syntax for the specification of properties that the code must satisfy. Hence, using this technique the security policies would be integrated inside the code structure among others specifications (e.g., functional requirements).

Example 3.5 The comment below specifies the file confinement policy of Figure 3.7.

```
\*@
  @ name   : file-confine
  @ aliases: read(f)  := (f:File).read()
  @          write(f) := (f:File).write(String t)
  @          new(f,d) := (f:File).<init>(String n, String d)
  @ states : q0 q1 q2
  @ start  : q0
  @ final  : q2
  @ trans  : q0 -- new(f,"/tmp") --> q1
  @          q0 -- new(f,d) --> q2 when d != "/tmp"
  @          q0 -- read(f) --> q2
  @          q0 -- write(f) --> q2
  @*\
```

The first part introduces the needed aliases. Then, the set of states $Q = \{\texttt{q0}, \texttt{q1}, \texttt{q2}\}$, the initial state `q0` and the set of final states $F = \{\texttt{q2}\}$ are declared. The automaton transitions close the definition.

Briefly, the policy $\texttt{file} - \texttt{confine}$ says that a program cannot access files created by others and can only create files in the `"/tmp"` directory.

Policy instantiation and enforcement. In the general case, each usage automaton admits an infinite number of possible instantiations (one for each valid combination of resources that can be mapped into its parameters). Moreover, different instantiations may share part of the resources they talk about or two different

instances may converge to the same state at run-time. Hence, we need to define an instantiation strategy that prevents unnecessary instances of the usage automaton from being created and optimizes the existing instances avoiding resources waste.

Our instantiation mechanism for usage automata uses configurations having the form:

$$\mathcal{Q} = \{\sigma_0 \mapsto Q_0, \ldots, \sigma_k \mapsto Q_k\}$$

where, for each $i \in \{0..k\}$, σ_i is a mapping from the parameters the automaton U to actual objects, while Q_i contains a subset of the states of U. Intuitively, each σ_i singles out a possible instantiation of U into a finite state automaton $A_U(\sigma_i)$, while Q_i represents the states reachable by $A_U(\sigma_i)$ upon the method trace seen so far.

The configuration $\mathcal{Q}$ of the mechanism is updated whenever a method is invoked at run-time. The formal specification is in Table 3.3, where with $\mathsf{R}(\mathcal{Q})$ we denote the set of objects occurring in $\mathcal{Q}$. The number of instantiations recorded by the mechanism grows as new objects are discovered at run-time. The distinguished objects $\#_1, \ldots, \#_p$ represent the objects before they actually appear in the trace.

The procedure of Table 3.3 shows how a sequence of method invocations is validated or rejected according to a usage automaton U. Roughly, step 1 represents the initialization of the usage automaton state. At this stage, the configuration consists of a single mapping pairing each parameter of U to a corresponding dummy resource $\#_i$.

Step 2 is the main loop processing the methods trace η. For each invocation $\mathtt{o.m}(\mathtt{o_1}, \ldots, \mathtt{o_n})$ a corresponding event is produced by through the defined aliases (a). Then, the current configuration $\mathcal{Q}$ is extended by instantiating the new mappings according to the freshly discovered resources appearing in the method invocation (b). Subsequently, the procedure select all the transitions of U that are compatible with the method under analysis and performs a corresponding update of the existing configurations (c).

Finally, step 3 verifies that no final state has been reached and returns *true*. Otherwise, if a violation occurred, returns *false*.

Example 3.6 Consider again the policy `file-confine(f,d)` of Figure 3.7 and the following methods trace:

$$\eta \; = \texttt{g.<init>("data", "/tmp")} \;\; \texttt{h.read()}$$

Applying the aliases defined by the policy, this trace is turned into the events trace:

$$\eta' \; = \texttt{new(g,"/tmp")} \;\; \texttt{read(h)}$$

The initial configuration of the monitor is:

$$\mathcal{Q} \;=\; \{\{\mathtt{f} \mapsto \#_1, \mathtt{d} \mapsto \#_2\} \mapsto \{\mathtt{q_0}\}\}$$

The first event of the trace under evaluation, i.e. `new(g,"/tmp")`, carries two resources, namely the object `g` and the static string `"/tmp"`. Moreover, the event is compatible with one of the transitions rooted in $\mathtt{q_0}$. Hence, the new configuration after the event is:

INPUT: a usage automaton U and a method calls trace η.
OUTPUT: *true* if η complies with the policy defined by U, *false* otherwise.

1. $\mathcal{Q} := \{\, \sigma \mapsto \{q_0\} \mid \forall x : \sigma(x) \in \{\#_1, \ldots, \#_p\} \,\}$, where q_0 is the initial state of U, $\#_1, \ldots, \#_p$ are distinguished objects, p is the number of parameters of U
2. while $\eta = \mathtt{o.m}(\mathtt{o_1}, ..., \mathtt{o_n})\ \eta'$ is not empty, do:
   (a) let $\mathtt{ev}(\mathtt{o'_1}, \ldots, \mathtt{o'_k})$ be the event abstracting from $\mathtt{o.m}(\mathtt{o_1}, \ldots, \mathtt{o_n})$
   (b) for all $\mathtt{j}$ such that $\mathtt{o'_j} \notin \mathsf{R}(\mathcal{Q})$, extend $\mathcal{Q}$ as follows. For all σ occurring in $\mathcal{Q}$ and for all mappings σ' from the parameters of U to $\mathsf{R}(\mathcal{Q}) \cup \{\mathtt{o'_1}, \ldots, \mathtt{o'_k}\} \cup \{\#_1, \ldots, \#_p\}$ such that, for all x, either $\sigma'(x) = \sigma(x)$ or $\sigma(x) = \#_j$:
   $$\mathcal{Q} := \mathcal{Q}\,[\sigma' \mapsto \mathcal{Q}(\sigma)]$$
   (c) let $step(q)$ be the set of states q' such that there exists an edge from q to q' with label $\mathtt{ev}(\mathtt{x_1}, ..., \mathtt{x_k})$ and guard g, where $\mathtt{ev}(\mathtt{x_1}, ..., \mathtt{x_k})\sigma_i = \mathtt{ev}(\mathtt{o'_1}, ..., \mathtt{o'_k})$ and $g\sigma_i$ is true. Let $step'(q) = \mathit{if}\ step(q) = \emptyset\ \mathit{then}\ \{q\}\ \mathit{else}\ step(q)$. Then, for all $(\sigma_i \mapsto Q_i) \in \mathcal{Q}$, update $\mathcal{Q}$ as:
   $$\mathcal{Q} := \mathcal{Q}\,[\sigma_i \mapsto \textstyle\bigcup_{q \in Q_i} step'(q)]$$
3. if Q_i contains no final state for all $(\sigma_i \mapsto Q_i) \in \mathcal{Q}$, then return *true* else *false*.

Table 3.3: Enforcement mechanism for policies defined through usage automata.

$$\mathcal{Q}' \;=\; \mathcal{Q} \cup \{\{\mathtt{f} \mapsto g, \mathtt{d} \mapsto \mathtt{"/tmp"}\} \mapsto \{\mathtt{q_1}\}\}$$

When the event `read(h)` is fired, the configuration update is applied to both the elements of $\mathcal{Q}'$. Since the event under analysis is compatible with a transition from the state $\mathtt{q_0}$ and introduces a new resource `h`, the monitor state evolves in the following way.

$$\mathcal{Q}'' \;=\; \mathcal{Q}' \cup \{\{\mathtt{f} \mapsto \mathtt{h}, \mathtt{d} \mapsto \#_2\} \mapsto \{\mathtt{q_2}\}\}$$

As $\mathtt{q_2}$ is final, the monitor detects a security violation and stops the execution.

Some optimization of the algorithm, omitted for brevity in Table 3.3, are possible. For instance, there is no need to allocate in step (b) a new instance $A_U(\sigma)$, unless $A_U(\sigma)$ can take a transition in step (c). Also, when an object `o` is garbage-collected, we can discard all the instantiations $A_U(\sigma)$ with $\mathtt{o} \in ran(\sigma)$; it suffices to record the states of $\mathcal{Q}(\sigma)$ in a special $\sigma^\dagger$ for disposed objects.

3.1.2 Policy sandbox

The possibility to define the local scope of a policy inside the normal sequence of instructions coding an application is a fundamental part of our framework. In general, a programmer should be able to apply a security policy to an arbitrary block of code that he wants to secure.

Here we survey the possible techniques for extending Java in order to integrate our security policies in the program instructions flow. We explain the solution we opt for and we show how it can be easily applied with no substantial changes to the normal structure of a program.

Native block abstractions. The Java libraries offer a convenient abstraction for blocks of executable code, that is the `java.lang.Runnable` interface. Mainly, a Runnable object consists of a `run()` method that simply contains a bunch of instructions. Typically, Runnable objects are used by Java Threads for defining the sequence of instructions that are scheduled during a parallel computation. However, Runnable objects are also used for many other purposes, e.g. for graphical events handling.

Another useful abstraction is provided by the `java.lang.reflect` package. This package contains the classes used for *reflection* in Java, that is the dynamic access to classes and data structure performed by a running application. In particular the instances of the class *Method* represent concrete methods. The standard way to force the execution of a Method object is through the method `invoke` in the following way:

```
Object result = M.invoke(target, arg1, ..., argN);
```

That is intended to behave like

```
Object result = target.m(arg1, ..., argN);
```

where `m` is the method that the object `M` represents.

There are several reasons why we prefer the former approach to the latter. Usability issues are predominant. Indeed, the Runnable representation is suitable not only for methods but also for common blocks of code. Moreover, the reflection mechanism requires a certain level of comprehension and awareness of the inner structure of the Java framework that is not required to every developer. Also the approach compatibility has been considered. Indeed, the Runnable interface is part of Java since the `1.0` version of the framework while the reflection package has been added later (version `1.2`).

Example 3.7 Consider now the following fragment of Java code.

```
try {
  File src = new File("private", "/etc");
```

```
  File dst = new File("public", "/tmp");
  // flush private data to public location
  String data = src.read();
  dst.write(data);
} catch(Exception e) {...}
```

The code for wrapping these instructions inside a Runnable object is:

```
new Runnable() {
  public void run() {
    try {
      File src = new File("private", "/etc");
      File dst = new File("public", "/tmp");
      // flush private data to public location
      String data = src.read();
      dst.write(data);
    } catch(Exception e) {...}
  }
}
```

The sandbox method. We use the method `sandbox` to define the scope of a security policy. This signature of `sandbox` is:

```
public static void sandbox(String P, Runnable C)
                           throws SecurityException
```

where the string `P` is the name identifying the usage policy (see Section 3.1.1) to apply on the execution of the code wrapped by `C`. Intuitively, executing a piece of code inside a sandbox can raise a `SecurityException`. If a violation of the policy is discovered, the sandbox interrupt the execution and throws the exception.

According to the procedure of Table 3.3, our monitor intercepts the method invocations during the execution of `run()`. If an invocation appears among the aliases defined by the policy, it is converted into a corresponding event and used to update the state of the monitor.

More policies can be easily composed by nesting the invocations to the `sandbox` method. A policy is activated when the execution enters the scope of a sandbox referring to it and deactivated when the block is left.

Example 3.8 Applying the `sandbox` method to the code of Example 3.7 we obtain

```
sandbox("policy-name", new Runnable() {
  public void run() {
    try {
```

```
      File src = new File("private", "/etc");
      File dst = new File("public", "/tmp");
      // flush private data to public location
      String data = src.read();
      dst.write(data);
    } catch(Exception e) {...}
  }
});
```

where `policy-name` is the identifier of a previously declared policy.

3.1.3 Security checks deployment

The last step for having a complete monitoring environment is the deployment of the instructions responsible for triggering the invocations to the security monitor.

Several mechanisms can be used to implement this operation. From a mere functional point of view, many of them are almost equivalent for our purposes. For instance, we could insert *local security checks* [36, 34] before each security-relevant method invocation. Local checks verify a security predicate and, if it is violated, halt the execution. However, local checks poorly integrate with the structure of the Java programs. In fact, we used *security proxies* that better fit with the object-oriented environment.

Security proxies. The approach used here exploits dynamically generated *proxies* for wrapping method calls. This mechanism is also implemented by JavaCloak [161] for extending the standard Java reflection support and improving the code mobility and re-usability.

Basically, a special *class loader* replaces the Java default one[1]. According to the Java specification [93], the JVM loads a new class when it is used for the first time during an execution. Thus, a special class loader can operate on the loaded classes and replace the security-relevant ones, i.e., those declaring some methods monitored by some of the declared policies, with proper proxies.

A dynamic proxy class is a class that implements a set of interfaces defined at runtime. The proxy class in used to encapsulate a concrete object and to create a further abstraction layer. Assuming that each security-relevant class implements a public interface, we load a proxy instead of the watched classes. We can see the working principle in the code below.

[1]Note that, since Java allows for having multiple class loaders, organised according to a specific hierarchy, we can still access to the standard class loader.

```
public class SecurityProxy implements InvocationHandler {
    private Object target;
    ...
    public Object invoke(Object proxy, Method M, Object[] args)
                         throws Throwable
    {
        Object result;
        if(check(target, M, args))              // monitor invocation
            result = M.invoke(target, args);    // method invocation
        else
            throw new SecurityException(M.toString());
    }
}
```

A proxy keeps a reference to the wrapped object `target`. When a method `m` of the original class is invoked, the wrapping proxy is delegated to satisfy the invocation. This is done through the method `invoke` of the `InvocationHandler` interface. The `invoke` parameters are: the target of the invocation, i.e. the proxy itself, the Method instance `M` representing the concrete method `m` and the arguments of the invocation.

Before dispatching the call to the actual class, the proxy updates the state of the security monitor through the function `check`. If the policy check succeeds, i.e. it returns *true*, the proxy complete the execution by performing the original invocation to `m` using its representation `M`. Otherwise, if an active policy is violated and the check fails, the proxy throws a security exception.

Since this technique exploits the Java object orientation, if offers some advantages. Firstly, the substitutability of proxies is guaranteed by the Java specification. Hence, we can use proxies instead of the classes they wrap exploiting a native feature of the language. In other words, if we have a proxy `P` for the class `C` we can write the following code

```
boolean b = p instanceof C;
C c = (C) p;
```

where `p` is an instance of `P`.

Another advantage of this approach is that it has a very limited impact on the original structure of a monitored program. Indeed, it suffices to include the special class loader for dynamically generating the proxies in a program. This means that the overhead due to the policy monitoring in terms of program dimensions is always a fixed amount.

However, the approach based on proxies has some limitations. Mainly, a proxy is usable and effective only if applied to a class implementing one or more interfaces. The shared interfaces are necessary for guaranteeing the substitutability properties of the proxy. This makes the approach not applicable to classes that implement no

interfaces. Also `final` classes, that are quite common among the Java APIs, can not be wrapped in this way.

These limitations make this technique not general enough for our purposes. Indeed, the security requirements of a program can involve each kind of classes or methods. Hence, we opt for a more general mechanism that is applicable to every Java program.

Bytecode instrumentation. A different way to force security checks in Java applications is through *bytecode rewriting*. A similar approach has been used by Kava [184, 185] for adding behavioural reflection to Java programs. Roughly, Kava allows for specifying some supplementary operations to execute just before and after accessing the members, e.g. methods and fields, of a class. The Kava instrumentation process ensures that these new pieces of code are correctly inserted in the instructions flow.

Example 3.9 Consider the following class.

```
class C {
  ...
  public D m(...) {
    // body of m
  }
}
```

The syntax for specifying the policy check code to be instrumented is the following.

```
class MetaC implements IMetaObject {
  ...
  public void beforeExecuteMethod(IExecutionContext c) {
    if(!check(c.getBase(), c.getMethod(), c.getParameters()))
      throw new SecurityException(...);
  }
}
```

Finally, the xml-based binding specification below drives the instrumentation process

```
<binding>
  <class>
    <classname>C</classname>
    <metaclass>MetaC</metaclass>
    <intercept>
      <execute>
        <method>m</method><parameters>*</parameters>
      </execute>
    </intercept>
```

```
  </class>
</binding>
```

Since we aim at creating an automatic instrumentation tailored for security controls, the Kava specification model is not satisfactory for our purposes. Indeed, our security framework always performs the same operations before a security-relevant invocation, that is an authorization request to the program monitor. Hence, specifying security checks using Kava would lead to long, repetitive descriptions for each method of each class where we want to insert a policy check.

Another serious issue posed by Kava derives from its applicability. As a matter of fact, some classes that we want to monitor can not be rewritten. For instance, this is the case of the classes using the *Java Native Interface*. Native methods are used in Java for accessing to routines and programs developed using other programming languages, e.g. C/C++. As they do not have a Java implementation, native methods cannot be rewritten. Many of the JRE libraries make use of native methods for accessing some functionality of the underlying platform. Needless to say that several of these methods can be involved in the monitoring process.

3.2 The Jalapa framework

The goal of this section is to present the *Jalapa* framework [22, 102]. The Jalapa project aims at enriching the standard Java application development with the support for specifying, verifying and enforcing local policies.

3.2.1 Framework structure

The Jalapa framework arises from the cooperation of few components. These components affect different stages of the life-cycle of a Java application, i.e. development, deployment and execution. We can distinguish four main parts composing the system.

- a bytecode rewriter, called *Jisel*, that modifies the application intermediate language by adding the facilities needed for dynamically enforcing local usage policies to Java programs.
- a *static analyser* of Java bytecode, that constructs an abstraction, namely a *history expression* [28], of the behaviour of a program.
- the *LocUsT* model checker [29] that reduces the infinite-state system given by the history expression to a finite one, and checks it against a set of policies.
- an Eclipse plugin that combines the previous items into a developer environment, with facilities for writing policies, sandboxing code, running the static analyses and compiling secured applications.

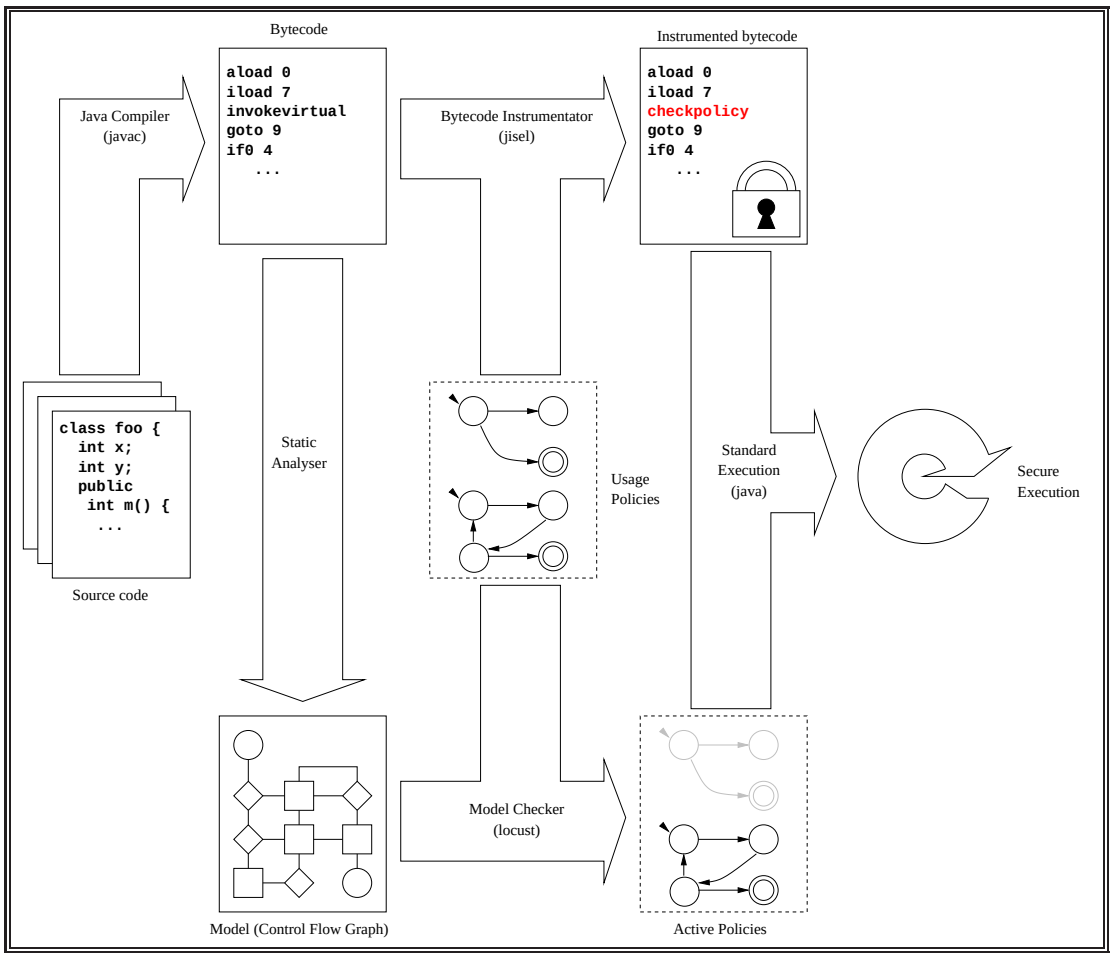


Figure 3.8: Application development in Jalapa.

Figure 3.8 is a graphic representation of the different steps during the life of an application in which the Jalapa components operate.

Basically, the application is coded using sandboxes for bounding the scope of local policies according to the syntax presented is Section 3.1. Moreover, the application developer specifies the structure of the usage policies referred to by the sandboxes.

The compiler output, i.e. the bytecode, is then used for both the static analysis and the instrumentation. From the one hand, the static analysis produces a suitable model of the application behaviour that can be model checked against the set of usage policies. The result of this process is a list of policies that have not passed the verification step and need to be actually enforced. On the other hand, the instrumentation modifies the bytecode by adding the runtime support for enabling the monitoring of local policies. The final step consists in executing the instrumented application deactivating the policies that do not appear in the model checker output.

3.2.2 The Jisel runtime environment

The first step towards implementing the Jisel runtime environment consists in intercepting the security-relevant operations of the program in hand, so to promptly block them before they violate a policy.

We use *bytecode rewriting* for inserting security checks before method invocations. However, differently from [185], we do not include the new code in the body of guarded methods. In fact, we use wrapping classes for encapsulating these methods in a secure context.

Bytecode instrumentation. The first step is detecting the set $\mathcal{M}$ of all the methods appearing in the aliases declarations of the policies. We inspect the bytecode, starting from the methods used in the aliases, and we then compute a sort of transitive closure, through a visit of the inheritance graph. Indeed, aliases can refer to abstract and interface methods and we need to find all the existing implementations.

We create a *wrapper* for each security-relevant class, i.e. a class declaring one or more security-relevant method. A wrapper $\texttt{W}_\texttt{C}$ for the class `C` declares exactly the same methods of `C`, implements all the interfaces of `C` and extends the superclass of `C`. This means that $\texttt{W}_\texttt{C}$ can replace `C` in any context. Indeed, it admits the same operations of `C`. The wrapper class $\texttt{W}_\texttt{C}$ has a single field, which will be assigned upon instantiation of `C`.

A method `m` of $\texttt{W}_\texttt{C}$ can be either monitored or not. If the corresponding method `m` of `C` does not belong to $\mathcal{M}$, then $\texttt{W}_\texttt{C}$`.m` simply calls `C.m`. Otherwise, $\texttt{W}_\texttt{C}$`.m` calls the `check` method that controls whether `C.m` can be actually executed without violating the active policies. A further step substitutes (the references to) the newly created classes for (the references to) the classes in the original program.

The instrumented code is deployed and linked to the Jisel runtime support, which contains the resources needed by the execution monitor. Note that our instrumentation produces a stand-alone application, requiring no custom JVM and no external components other than the Jisel library. The Jisel preprocessor takes as input the file containing the needed policies, the directory where the class files are located, and the directory where the instrumented class files will be written.

The `check` and `sandbox` methods are provided by the static class `PolicyPool`. The PolicyPool class implements the enforcement mechanism in Table 3.3 and controls the whole execution monitoring process.

Runtime monitoring. The resources handled by the PolicyPool are: the definitions table of the instrumented policies, a stack containing the references to the active policies and a set of policy automata. Whenever the execution flow enters a sandbox, the corresponding policy is retrieved from the definitions table and instantiated to the initial configuration automaton. A reference to the policy is pushed on the stack and its automaton added to the set of existing ones. Symmetrically, when

a sandbox is left, the policy on the top of the stack is removed and all its automata deleted. The code implementing the sandbox is:

```
public static void sandbox(String policy, Runnable code)
                              throws Throwable
{
  Throwable ex = null;
  try {
    activate(policy);
    code.run();
  }
  catch(Throwable e) {ex = e;}
  finally {
    deactivate(policy);
    if(ex != null) throw ex;
  }
}
```

where the `activate` and `deactivate` work as previously explained in Section 3.1.

Policy automata have a configuration representing their state as described in Section 3.1.1. These configurations contain references to the resources involved in the monitoring process. In order to avoid interferences with the Java garbage collector we use *weak references* [80]. As a matter of fact, standard references would prevent the garbage collector from disposing unused objects only pointed by the PolicyPool data structures, so potentially leading to memory exhaustion. An object only pointed by weak references is considered unreachable, so it may be disposed by the garbage collector. When the PolicyPool notices that a resource has been garbage collected, it destroys the policy instances that are no more necessary.

The method `check` has the same syntax given in Section 3.1.2. The only difference resides in its usage: since it is a static member of the PolicyPool class it is invoked with the instruction `PolicyPool.check(...)`. The result of `check()` is true if and only if no policy automaton reaches an offending state. If so, the method invocation is authorized and the wrapper forwards it to the actual class; otherwise, a `SecurityException` is thrown.

Executing secured applications. As we said above, an application instrumented with our method is a standalone program that needs no external components for monitoring. Working on the application bytecode, rather than on its source code, the instrumentation routine can also be applied to Java libraries and JAR archives in general. However, note that, without sandboxes inside the program code, we can only apply policies with a global scope.

To run the instrumented program, one must supply the set of policies to be enforced, besides the inputs of the original program. Policies can belong to two

different categories: *global* and *local*. If a policy is declared global its scope is applied to the whole execution even if no corresponding sandbox invocation exists. In this case, the PolicyPool instantiates the global policy before starting the program execution. Instead, the policies of the second type, i.e. local, behave in the standard way according to the position of the sandboxes. Finally, the policies that are not in one of the two groups are ignored by the runtime monitor.

We use the JVM system properties for declaring the two lists of policies in the command line syntax as follows.

```
# java [-Dcheck.global=<pols>] [-Dcheck.local=<pols>] <app> [par1 ... parN]
```

where `<pols>` is a list of names and `<app>` is the application name.

3.2.3 Static analysis and verification

While the run-time enforcement mechanisms of Jisel ensure that policies are never violated at run-time, they perform checks at each method invocation, imposing some overhead on the code running inside a sandbox. In order to mitigate this overhead, we verify programs to detect those policies that are always respected in all the possible executions of the sandboxed code.

Verification of usage policies. For those policies that may fail, our technique finds (an over-approximation of) the set of method calls that may lead to violations. By exploiting this information, the run-time enforcement can be optimized, since it is now safe to skip some run-time checks. Note that, in the most general case, the Java source code could be unavailable, yet one might still want to optimize its execution. To this aim, we perform our verification on the Java bytecode.

Our verification technique consists of two phases, briefly described below.

- first, we extract from the bytecode a *control flow graph* (CFG), and we transform it into a history expression
- then, we model-check the history expression against the usage policies enforced by the sandboxes used in the program.

The syntax of history expressions resembles that of the *basic process algebra* (BPA) [33]. Basically, an history expression H can be empty (ε), a variable (h), an action ($\alpha(\bar{r})$ where $\bar{r}$ is a finite vector of resources), a non-deterministic choice ($H + H'$), a sequence ($H \cdot H'$), a policy framing ($\varphi[H]$), a recursion ($\mu h.H$) or a resource creation ($\nu r.H$). We refer the interested reader to [28].

We now show an example where the verification technique can be exploited to remove unneeded checks.

Example 3.10 Consider the following code fragment:

```
PolicyPool.sandbox("file-confine", new Runnable() {
  public void run() {
    File f = new File("log", "/tmp");
    String s = f.read();
    buffer.add(s);
    for(int i = 0; i < buffer.size(); i++)
      f.write(buffer.get(i));
  }
});
```

Roughly, this block creates a new file object, with path `log/tmp`, and writes into it the content of the `buffer` data structure. In doing that, the procedure also rewrites the previous file content after the new one. The static analyser extracts the control flow graph, and transforms it into the following history expression:

$$H_{\texttt{log}} = \texttt{new(f, "/tmp")} \cdot \texttt{read(f)} \cdot \mu h.\,(\texttt{write(f)} \cdot h + \varepsilon)$$

where $\cdot$, $+$ and μ are the operators presented above and the events `new`, `read` and `write` are those defined by the aliases of the policy `file-confine` (see Section 3.1.1). Intuitively, $H_{\texttt{log}}$ represents all the traces creating a new file `f` in the `"/tmp"` directory, reading its content and then repeatedly writing on it.

We then check $H_{\texttt{log}}$ against the `"file-confine"` policy using the model checker of Jalapa, and discover that no possible trace violates the policy. Therefore, we can safely remove the sandbox, and directly execute the code, so improving its performance.

The static analyser. The *control flow graph* (CFG) of a program is a static-time data structure that represents all the possible run-time control flows. In particular, we are interested in constructing a CFG the paths of which describe the possible sequences of method calls. This construction is the basis of many interprocedural analyses, and a large amount of algorithms have been developed, with different tradeoffs between complexity and precision [96, 141]. The approximation provided by CFGs is *safe*, in the sense that each actual execution flow is represented by a path in the CFG. Yet, some paths may exist which do not correspond to any actual execution. A typical source of approximation is dynamic dispatching. When a program invokes a method on an object `o`, the run-time environment chooses among the various implementations of that method. The decision is not based on the declared type of `o`, but on the actual class `o` belongs to, which is unpredictable at static time. To be safe, CFGs over-approximate the set of methods that can be invoked at each program point. Similarly, CFGs approximate the data flow, e.g., by relating object creation (`new`) to the location of their uses (method invocations). For each method invocation occurring in the program, say `m(x)`, the CFG defines a set of all the possible sources of the object denoted by x, i.e. which `new` could have

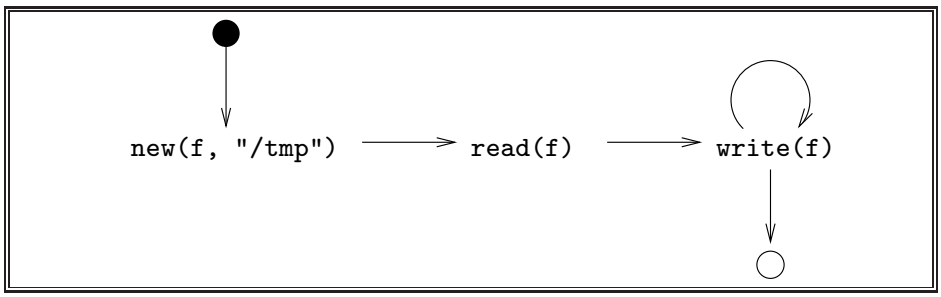


Figure 3.9: The event graph producing the history expression $H_{\texttt{log}}$.

created `x`. Again, this is a safe approximation in the sense that this is a superset of the actual run-time behaviour.

For each policy for which the original program defines a sandbox, the CFG extracted at the previous step is transformed into an *event graph*. This operation involves substituting events for method signatures, according to the aliases defined in the policy, and suitably collapsing all the other nodes of the graph.

Finally, the event graph is transformed into a history expression. This is done through a variant of the classical state-elimination algorithm for finite state automata [46].

Example 3.11 Consider again the procedure shown in Example 3.10. The history expression $H_{\texttt{log}} = H_{\texttt{log}} = \texttt{new(f, "/tmp")} \cdot \texttt{read(f)} \cdot \mu h.\,(\texttt{write(f)} \cdot h + \varepsilon)$ is obtained from the event graph depicted in Figure 3.9.

One of the main issues in converting CFGs to history expressions is to correctly track objects. For instance, in Figure 3.9 it is important to detect that `f` always denotes the same object, so to express this information in the history expression without losing precision. When this is not possible, e.g., because `f = new File(...)` occurs in many places in the code, we need to use a more approximated history expression. This is done by exploiting non-deterministic choice ($+$).

The LocUsT model checker. The final phase of the analysis consists in model-checking history expressions against usage policies. To do that, history expressions are transformed first into Basic Process Algebras (BPAs, [33]), so that we can apply standard model-checking techniques [87].

The *LocUsT* model-checker has been implemented as a Haskell program running in polynomial time in the size of the history expression extracted from the event graph. Full details about this technique and LocUsT can be found in [27, 29].

Briefly, the model-checking algorithm verifies that no trace of the BPA is also a trace recognized as offending by the policy. To do that, it checks the emptiness of the pushdown automaton resulting from the conjunction of the BPA and the policy (which denotes the unwanted traces). The transformation into BPA preserves the

validity of the approximation, i.e. the traces of the BPA respect the same policies as those of the history expression.

The final result of this process is a list of security policies that still need to be dynamically enforced. As seen above, we use this list to set the `check.local` and `check.global` system properties when running an instrumented application. Note that these lists can also be instrumented together with the monitoring support in order to apply them to every execution of the program with no need for external commands.

3.3 On-device monitor inlining

In this section we move to a specific platform context, i.e. *mobile devices*. Mobile devices, e.g. smart phones and personal digital assistants, are becoming more popular day by day. As their computational capability and memory capacity are growing rapidly, users start to exploit them for many purposes other than just making and receiving phone calls. Moreover, they also handle many highly critical resources that need to be protected, e.g. personal data and phone credit.

The Java environment for mobile devices, namely *Java ME*, has received major attention and several security studies have been presented about it, e.g. by Kolsi and Virtanen [108] and Debbabi et al. in [72, 73, 74]. Many possible threats have been pointed out and, in particular, some architectural vulnerabilities have been identified.

Natively, the Java ME platform provides some basic security support for Java ME applications, i.e. MIDlets. However, this mechanism does not provide a real protection against malicious programs. In practice, the standard Java ME approach consists of associating each MIDlet to a proper security domain. Different security domains have different privileges and can access to different sets of system resources. The security domain for a new installation depends on the signature of the MIDlet. For instance, if the MIDlet is signed by a trusted certification authority it is placed in an high level domain. Otherwise, if the MIDled is not signed, it receives no privileges. Whenever a running MIDlet tries to access a resource that is not in its security domain, the user is prompted with a request for an explicit permission.

This approach to application security is not satisfactory for several reasons. Mainly, it charges the users with the responsibility of deciding whether a certain action is safe or not. Needless to say that the mobile devices users are rarely aware about the problems of security. Hence, we propose a different technique for securing Java ME applications through a runtime monitor.

3.3.1 Application monitoring on mobile devices

The proposed monitoring environment works by exploiting two components: a central monitoring entity called *policy decision point* (PDP) and one or more sources

of security events called *policy enforcement points* (PEPs). The behaviour of the PDP is quite simple. Roughly, it receives a signal from a PEP, it evaluates the event against the current security state and it answers allowing or prohibiting the operation. Instead, the PEPs are responsible for intercepting the system events and applying the PDP's response.

In the remainder we detail the parts of our system and we explain their behaviour and interactions.

Policies and actions. For the specification of security policies we adopt the *ConSpec* language. ConSpec has been specifically designed for resource limited platforms as part of the *Security for Software and Services for Mobile Systems* (S3MS) project [164] [81]. ConSpec is inspired by Erlingsson and Schneider's PSLang [86] and its formal semantics is given in terms of Schneider's security automata [168]. A complete dissertation about ConSpec and its features can be found in [6] and [7].

In this context we do not use local policies as we did in Section 3.1 for standard Java application. This choice is motivated by the different working assumptions of the current scenario. Indeed, here we want to prevent an entire, possibly untrusted, application from misusing the system resources. Hence, we do not assume a MIDlet to contain any security mechanism such as a policy sandbox. In other words, every policy that we apply to a running MIDlet must be considered "global" with respect to the entire application.

Another important issue is that here we focus on a predefined set of operations to be monitored. In particular we aim at defining security policies over the sequences of invocations to the Java ME system libraries. As a matter of fact, Java ME provides a quite rich set of functionalities for handling the resources of a mobile device and mediating the access to them. Furthermore, the inner structure a MIDlet is commonly unknown or even obfuscated [59]. Hence, allowing for a customisable set of monitored actions would produce a system overhead with few or even no benefits for the expressiveness of the policy formalism.

We now informally describe the structure of a ConSpec policy. A ConSpec policy consists of a set of rules. Each rule has a *scope* defining its validity extension. There are four possible scopes: *object, session, multi-session* and *global*. The object scope indicates that the policy rule refers to a specific instantiation of an object. Session scope says that the rule applies to the entire session of a MIDlet execution. Policy rules with a multi-session scope are enforced on multiple execution of the same MIDlet. Finally, global rules are enforced on all the MIDlets running on the mobile device. In other words, rules with a global scope are used to handle a shared security state that is affected by all the MIDlets and their interactions. The security state of a policy is defined by means of a set of variables.

Each rule defines the authorization conditions that must be satisfied before allowing a MIDlet to execute a certain security-relevant action and the instructions for updating the security state. A condition is a boolean guard, expressed accord-

```
1 SCOPE session
2 SECURITY STATE
3 int smsNo = 0;

4 BEFORE javax.microedition.io.Connector.open(String url)
5 PERFORM
6 (url.startsWith("sms") && url.getNumber().startsWith("+39")) -> skip

7 BEFORE javax.wireless.messaging.MessageConnection.send(Message msg)
8 PERFORM
9 (smsNo < N) -> smsNo++;
```

Table 3.4: An example of ConSpec security policy.

ing to the Java syntax. Similarly, the state updating is defined through the Java commands and a special keyword `skip` for no actions.

Table 3.4 shows an example of ConSpec policy. The scope of the policy is session (line 1), hence the policy is enforced on each MIDlet instance and its state (lines 2 and 3) reinitialises at every new execution. The first clause of the policy (lines 4-6), authorizes the MIDlets to open a connection using the protocol "sms" only if the phone number starts with "+39". The second clause of the policy (lines 7-9) concerns the method `javax.wireless.messaging.MessageConnection.send`, used for sending short text messages, and states that before the execution of the method, the value of the variable `smsNo` must be less than the constant N. If this condition is satisfied, the execution of the method is allowed, and the value of `smsNo` is incremented. Since ConSpec uses a *default: deny* approach, the actions that do not fit in these rules are prohibited. Hence, this policy allows each MIDlet to send no more than `N` messages to numbers with prefix "+39".

Moreover, the enforcement of a policy could also require that some system information is retrieved from the device. For this purpose, ConSpec is integrated with a rich class of specific operations for retrieving system information. They include: the types of available network interfaces (e.g. WiFi and Bluetooth), the battery level, the system time and many others. The policy rules can use this information in defining guards that, for instance, prevent some specific applications from being executed when the battery level is below a certain threshold.

Monitor structure. From an architectural point of view, the runtime monitor framework has been integrated with the Java ME architecture following the *reference monitor* model of [168]. It exploits the PEP component to intercept the invocations to security-relevant methods before they are actually executed, asks the PDP for the permission to proceed and enforces the PDP response. The PEPs are responsible

for monitoring the MIDlet during its execution. In particular, it intercepts all the security-relevant actions that the MIDlet tries to perform on the underlying mobile phone. Clearly, in order to be effective, the PEPs must catch every instance of the actions they are responsible for.

Instead, the PDP reads the security policy and handles the decision process. In practice, it is responsible for evaluating whether a given action is permitted in the current state by the policy. When stimulated by an invocation from a PEP, it exploits the *policy information service* component to manage the policy state. If the policy evaluation requires pieces of information from the device, they are retrieved through the *system information service*.

It is important to note that this architecture for events monitoring is general with respect to all the possible PEP implementations. Indeed, the PDP receives external signals through a connection channel in a way that is independent from the signal source. This technique makes the approach general and reusable in many different contexts and also in presence of heterogeneous PEPs, i.e. PEPs implemented and deployed in different ways. As an example consider the two monitoring architectures shown in Figure 3.10 and 3.11. Figure 3.10 depicts the monitoring architecture where PEPs have been added to the functionality of a customised Java virtual machine [132, 79]. It is straightforward that the PDP structure, on the right hand side, keeps unchanged with respect to the system deploying the PEPs through in-lining of Figure 3.11.

3.3.2 Bytecode in-lining

Hereafter we discuss our bytecode in-lining procedure for Java MIDlets. The in-lining process is an instrumentation step performed on the MIDlets bytecode before being executed by the Java ME virtual machine, called KVM. There are several differences with the instrumentation technique that we presented in Section 3.1. Mainly, the in-lining result is not a standalone application. Indeed, it can be executed only in presence of a working PDP. Moreover, we must consider the platform constraints and the reduced capabilities of Java ME. For instance, here we do not need to cope with dynamic class loading. However, as efficiency is crucial on devices having limited resources, we aim at implementing a fast and lightweight monitoring environment.

Instrumentation time. Basically, the in-lining process is performed by an *in-liner*. The in-liner is responsible for instrumenting the MIDlet bytecode before it can be executed. A possible approach would consist in applying the instrumentation before the first execution. However, as we will see later in this section, the instrumentation process requires a certain amount of time that is clearly noticed by the users. Even though this step needs to be performed only once, the first execution of a MIDlet would be substantially delayed. This would produce an annoying effect to

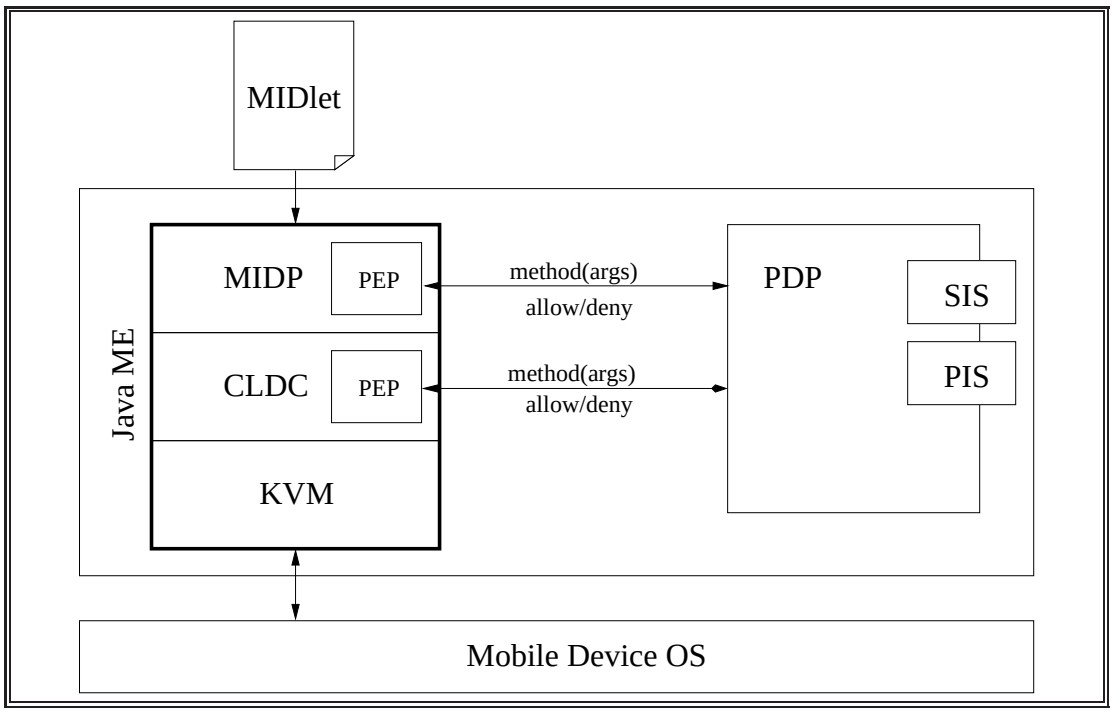


Figure 3.10: Runtime monitoring architecture using a customised JVM.

the user and could cause problem to MIDlets that are time dependent (for instance, imagine the first execution of an alarm clock MIDlet).

Instead, our solution consists in inserting the in-liner execution, called instrumentation time, between the retrieval of a MIDlet and its installation. As a matter of fact, these two steps are almost always executed in sequence. Usually, MIDlets are retrieved from some provider, e.g. downloaded from an application store or received from another device through a bluetooth connection. MIDlets are packaged in Java archives, i.e. JAR files, containing their class files and resources, e.g. pictures and sound. The installation process takes a JAR file, verifies its signature and deploys it according to the platform specifications, e.g. it could simply copy the JAR into the KVM working directory.

Before the installation of a JAR file, our in-liner instruments it with the security code and creates a new installable package. Then, the new package is installed in place of the original one. Pragmatically, we aim at preserving the system usability by including this step in the installation process. Indeed, the users is typically aware about the fact that program installation is time consuming. Hence, the instrumentation is included among the standard operations and exposed to the user as shown in Figure 3.12.

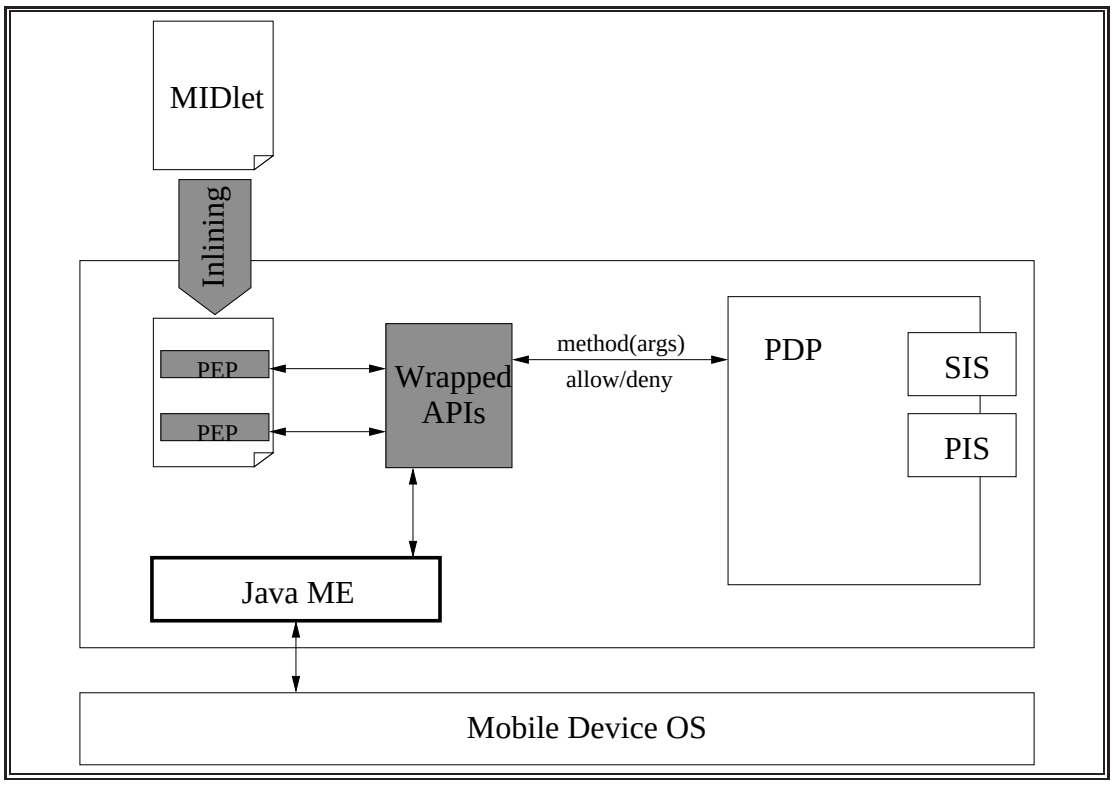


Figure 3.11: Runtime monitoring architecture using the in-lining approach.

Instrumentation procedure. The in-liner starts by loading the list of the security-relevant API calls, i.e. the list of Java ME methods to be monitored. This list can be optimised by synchronizing the in-liner with the settings of the PDP. Otherwise, we can also decide to monitor all the API invocations. In our system, we conventionally decide to have a predefined subset of methods that can be monitored, i.e. the policy alphabet. For instance, we can include in the alphabet several classes of the `javax.microedition.io` package, responsible for many networking functionalities, and none of the `javax.microedition.lcdui`, handling graphics components. In this way, we assume networking activities to be potentially involved in the security evaluation, while we do not care about the access to graphics facilities. For each class involved in this process we have a corresponding wrapper. In this way, wrappers redefine part of the Java ME APIs, from which we call them *wrapped APIs*. Note that, differently from the wrappers of Section 3.2, these classes are statically defined. Hence, the in-liner has a pre-compiled package containing the wrapped APIs instead of automatically producing them at instrumentation time.

A further advantage deriving from the static wrappers is that we can use the original classes exceptions for making our enforcement more transparent. As a matter of fact, almost every security-relevant method can throw some exception. These

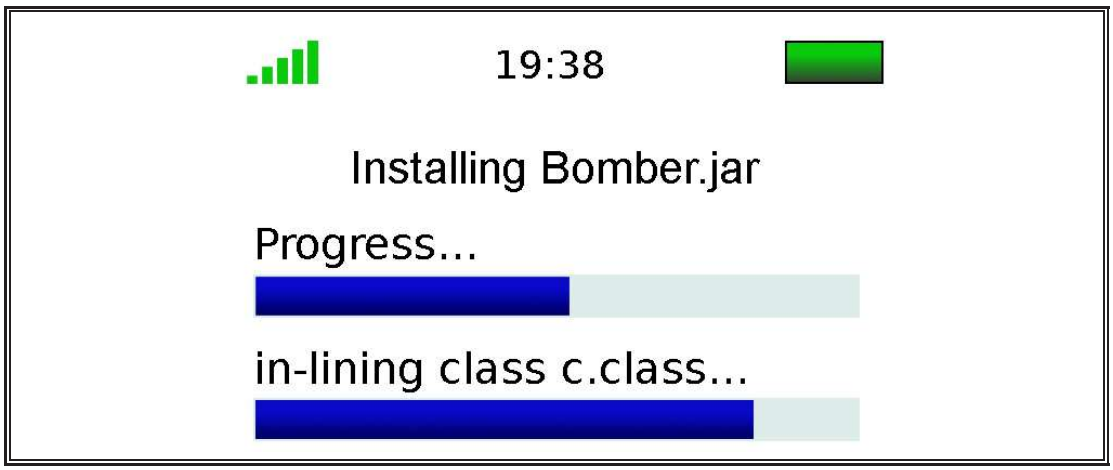


Figure 3.12: The instrumentation step of a MIDlet.

exceptions are declared in the method signatures and programs should handle them. In this way, we created wrappers that, in case of a security violation, do not raise a `SecurityException` that in many cases would lead to a system failure killing the application. Indeed, when a violation occurs, our wrappers throw an exception having a type that is compatible with the original API invocation. For instance, the wrapper for the method `Connector.open(...)` throws a `IOException`. Consequently, if a MIDlet implements a correct exception handling procedure it can continue its execution even after such an exception. Needless to say, this possibility can contribute for the overall system usability.

The second phase of the in-liner is inspecting the MIDlet bytecode sequence looking for corresponding invocation operational code. These positions are replaced by the invocations to the wrapper code implementing the PEP operations. Since the wrapped APIs have the same signature as the original methods, a simple rewriting of the invoked method name is sufficient. The PEP mainly performs three operations: (i) it converts the current method call and its parameters into a PDP message, (ii) it sends the message to the PDP using an internal, inter-process connection and (iii) it receives and enforces the answer of the PDP. The first two steps are straightforward. Indeed, at runtime the n parameters of a method invocation, including the object `this` for non-static calls, are stored in the top n positions of the operand stack. Hence, the wrapper receives the n parameters, prepares the message and sends it. If the PDP answers with an "allow" message, then the wrapper calls the original method, otherwise it raises an exception according to what we said above. Before leaving its scope, and returning to the code of the application, the PEP performs a second check on the return value of the API invocation.

Figure 3.13 shows the result of instrumenting a piece of bytecode. Imagine that the k-th instruction is a monitored method invocation. The in-liner replaces it with the invocation to a proper wrapper that executes the PEP code before and after the

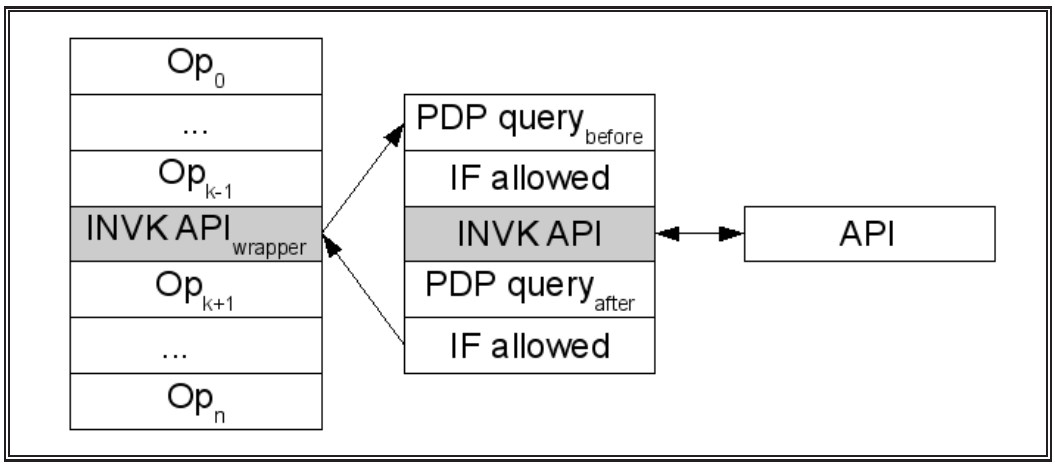


Figure 3.13: The in-lining of a single API call in a bytecode sequence

original invocation.

Remarks and open issues. Part of the PEP execution exchanges messages with the PDP through a local network interface. This operation represents a possible threat for the whole monitoring system. Indeed, it creates the conditions for a *man in the middle attack* [176]. An attacker could try to delete or inject some message from/to the PDP. Then, the communication between a PEP and the PDP should be secured, e.g. via encryption. These techniques should be considered in order to find a reasonable compromise with the performances of the system.

The current PEP strategy allows the in-lined application for running only if a PDP is working properly. Indeed, whenever the PDP does not answer to a request, the PEP reacts as if a "deny" command was received. In particular, when a MIDlet starts its actual execution a signal is sent to the PDP. This is obtained since we always instrument the `StartApp` method, that is the entry point of a MIDlet execution, both when it is explicitly required by the current policy or not. If the PDP does not reply, i.e. it is not running or has been corrupted, the PEP handles the lack of information as a negative response and aborts the current execution. Moreover, we also wrap the methods implementing the state change of a MIDlet, i.e. *Paused*, *Active* and *Destroyed*. These events are necessary for managing the policies scope and for a variety of policies of interest, e.g. applications black or white list.

Other important issues arise from the Java ME framework. Indeed, we observe that the modifications to the structure of a MIDlet invalidate its signature, if any. Clearly this means that, on the current devices, an in-lined MIDlet loses all the access privileges provided to its original, unmodified version. However, since our framework is intended to be an alternative to the certificate-based one, this is not a drawback of our approach. Furthermore, our system seems particularly appropriate for the current mobile software scenario. Indeed many MIDlets are produced by

companies and developers that for various reasons, e.g. high costs, do not apply a proper signature to their applications. Using our security model, users can install and execute unsigned, well-behaving MIDlets with no limitations on their usability.

Taking into account the consequences of rewriting (part of) MIDlets is another crucial issue. Indeed, as Java is not provided with a formal semantics, actually we can not prove formally that the segments of code added for each PEP do not introduce any unpredicted behaviour. However, since every PEP is composed by few, atomic instructions not interacting with the other parts of the MIDlet, we are confident that in-lined MIDlets preserve their integrity (remember that an in-lined wrapper takes the same arguments, returns the same value and, possibly, throws the same exceptions as the corresponding original method).

3.3.3 System implementation

In this section we detail our prototype implementation and we give several experimental results. The performances evaluation has been obtained by comparing our system with a similar one using a customised KVM [132] instead of bytecode instrumentation.

Platform and tools. The implementation of the prototype of the in-lining approach has been carried out using different mobile devices. In particular the whole architecture has been installed and tested on the Nokia phones E61 and N78. Nokia E61 works with Symbian v9.1 S60 3rd edition operating system while Nokia N78 runs Symbian v9.3 FP2. Both of them support the Java 2 Micro Edition Mobile Information Device Profile (MIDP) v2.0 (JSR 118) [105]. As previously explained, the adoption of the in-lining approach does not require the modification of any components on the mobile phone, such as the operating system or the Java ME platform, but it is sufficient to install additional software: the in-liner and the PDP .

The in-liner component of the prototype has been entirely implemented in Java ME. Considering the reduced computational capabilities and the restrictions on memory usage posed by most of the current mobile devices, the in-lining process seems to be on the edge of what is actually feasible on these platforms. Indeed, it handles compressed Java archives, reads and modifies class files, changes the inner structure of the target MIDlet (and the corresponding JAD description file) and, finally, builds a new executable MIDlet.

The files compression issue needs a further comment. As a matter of fact, the compression/decompression algorithms require such a computational effort that the KVM prefers to delegate them to external, system code. The Java Standard and Java Enterprise obtain this by exploiting the Java Native Interface (JNI). Unfortunately, this approach is infeasible for Java ME, that is provided with no such APIs, and it has been necessary to create a light-weight, fully Java-implemented compression library. Another complication arises from class files. Actually, class files can exceed the maximum memory size available for the running Java ME applications. This

problem is solvable by implementing a partial-representation mechanism that can work on classes without having their complete structure in memory.

The last step of the in-lining process is the MIDlet re-assemblage. In doing this, we add the wrapping classes to the original code of the application. Since the APIs to be wrapped are a well-known, finite set, we preferred to create the corresponding classes in advance. This solution requires a few KiloBytes of memory space but relieves the in-liner from the burden creating these classes at runtime. Again we have to use our compression library to create a valid JAR file and, in addition, we modify the Java description file (JAD) changing some entries, e.g. the archive size and the application entry point. The result of the procedure is a new JAR archive including the instrumented MIDlet, ready to be installed on the mobile device.

The policy decision point has been implemented in C++ for Symbian v9.1 and runs also on later Symbian OS versions. The C++ implementation has been chosen mainly for efficiency reasons. As a matter of fact, since the PDP code is executed twice for each security relevant method invoked by the MIDlet, a inefficient implementation could introduce a considerable overhead in the MIDlet execution time. The PDP is a daemon, it is started at device initialization time and it is always active on the mobile device. Once activated, the PDP daemon reads the security policy, builds the policy internal representation that is used to test the actions against the security policy, and suspends itself waiting for an invocation from a PEP.

The communication between the PDP daemon and the PEPs embedded in the MIDlets has been implemented through a local socket. The security policy itself prevents MIDlets from communicating directly with the PDP to interfere with the policy evaluation process. This is possible because any connection that an in-lined MIDlet tries to perform should be first authorized by the PDP. Moreover, if the PDP is unavailable, the PEP automatically denies any action that the MIDlet tries to execute. An advantage of this choice is that all the MIDlets that are executed on the mobile device share the same PDP. This allows to naturally implement global security policies, i.e. policies that take into account the actions performed by all the MIDlets running on the device. As an example, imagine a security policy stating that no more than N sms messages can be sent every day. In this case, the PDP should take into account the sms messages that have been sent by all the MIDlets. Similarly for a policy that states that a MIDlet A can be executed only if another MIDlet B has not been executed yet.

Performances. Monitoring the running MIDlets introduces an execution overhead mainly due to the evaluation of the security policy for each action performed. The impact of the policy evaluation on the execution time depends on several factors, such as the complexity of the security policy and the actions executed by the MIDlet. As a matter of fact, the security policy evaluation time depends on the number of rules concerning the current action and on the complexity of the predicates to be evaluated for these rules. Moreover, the impact of the monitoring on

the MIDlet execution time increases with the rate of security relevant actions it performs.

To obtain a realistic evaluation of the overhead, we carried out our test using a real case MIDlet, the Proteus PicoBrowser [159], an open-source Java ME Internet browser. We slightly modified the original version of Proteus by adding a timer to measure the execution time. When the browser is asked for loading a new HTML page, the timer starts. Then Proteus retrieves the required file, interprets its content and shows it on the screen. At this point, the timer stops returning the elapsed time. The security-relevant method that we decided to monitor is the `javax.microedition.io.Connector.open`, i.e., the API responsible for creating new connections. In our experiments, we enforced distinct policies each containing a different number of rules. All these rules concern the `open` method, and hence they are all evaluated each time that the PDP is invoked. Thus, from the overhead point of view, this is the worst case. In fact, the computations required for analysing the policy when a security-relevant action is executed strictly depends on the number of rules referring to that particular action.

We executed the test MIDlet on both the prototypes, i.e. running its in-lined version on a standard KVM and the original version on the customised virtual machine. We repeated the same tasks, under the same conditions, for both the original and the monitored prototype. To avoid the influence of network-dependent delays, we accessed a local page, i.e. a page stored on the mobile device itself. In order to minimize the irrelevant computation we used light html document containing only a short text. In this sense, the results of our test must be interpreted as the maximum cost due to the monitoring process during a page loading. Indeed, loading a bigger file from the network would lengthen the total execution time reducing the percentage of impact of the monitoring operations. We repeated our measurement several times and we computed the average values.

Figure 3.14 shows the results of the test we performed. The top chart refers to the in-lining approach. We installed the required components, i.e. the in-liner and the PDP, on a Nokia N78 mobile phone and we executed the test MIDlet under different conditions. The first value of the chart, represented by the white bar, refers to the execution of the original MIDlet. The second is the time for the in-lined MIDlet guarded with a single-rule policy and the third is monitored using a policy with 5 rules. The number of rules has been considered in our experiments since the PDP needs to verify more conditions before deciding whether an operation is granted or not. This, in principle, could lead to a longer computation and, consequently, to more noticeable delays. We opted for these two policy sizes because we observed that many policies of interest can be written in less than 5 rules. For instance, the policy of Table 3.4 is the composition of two, single rule policies defined over two different methods.

We observe that in the case of a policy with 1 rule the overhead is about the 12% of the whole execution time. This delay is due to the presence of an extra process in the system, i.e. the PDP, and to its interactions with the monitored application.

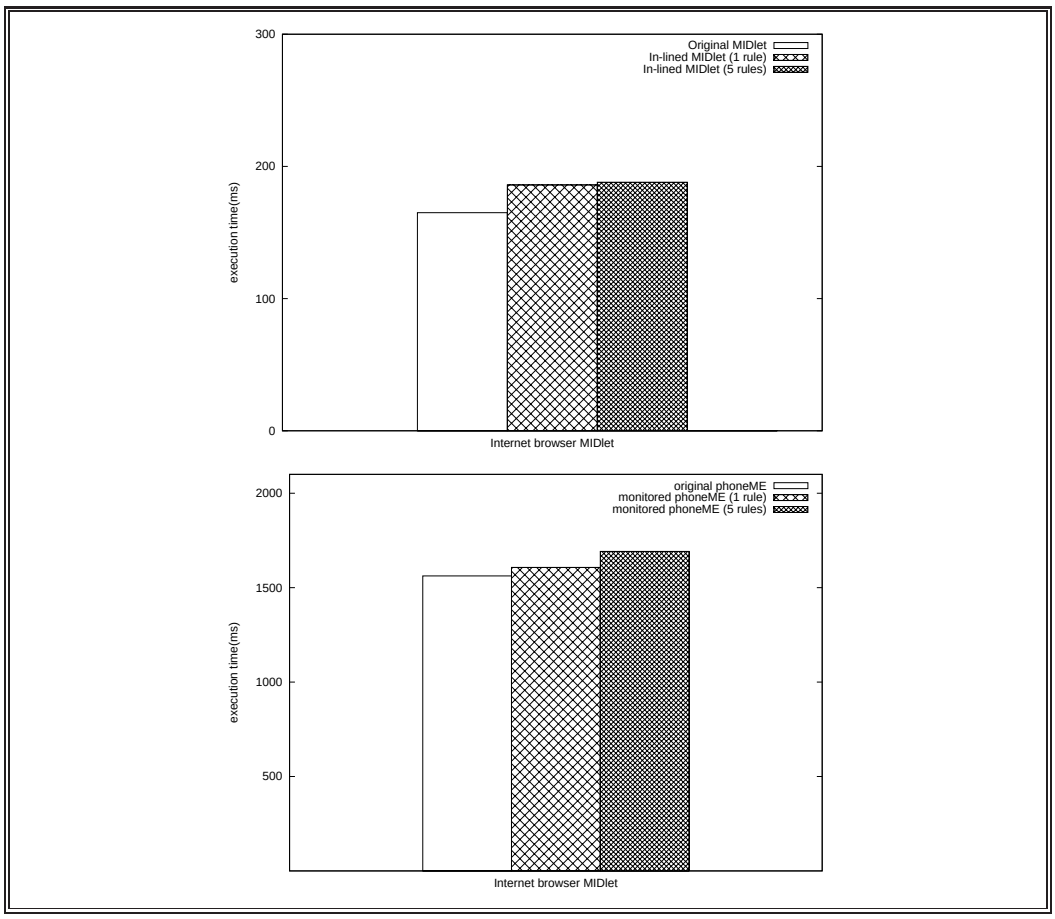


Figure 3.14: Monitoring overhead performances comparison.

When increasing to 5 the number of rules, we obtain a further 1% performances overhead. These results outline that our monitoring mechanism allows for applying very expressive policies with moderate consequences on the execution time.

The bottom chart of Figure 3.14 reports the results concerning the experiment that uses a customised KVM. The security enhanced KVM has been installed on an HTC Universal smartphone QTEK 9000 running Linix Openmoko OS [143]. Since this device has a different hardware profile with respect to the N78, we can not make a direct comparison among execution times. However, we can compare and analyse the percentage of execution time due to the monitoring system. Also in this case, the first result, i.e. the white bar, refers to the execution of the MIDlet on the original KVM, the second bar refers to the execution of the MIDlet on the modified KVM and a policy with one rule and the third bar refers to the execution of the MIDlet using a policy with 5 rules. We can observe that the overhead in the case of a policy with 1 rule is about 3%, while in the case of a policy with 5 rules is about 8%.

From these observations we deduce that, as expected, a customised KVM performs better than the instrumented MIDlet. This difference derives from the two different implementation of the PEPs: C++ for the customised KVM and Java for the in-lined MIDlet. Clearly, Java implemented operations are compiled into bytecode instructions and then interpreted by the KVM while the C++ routines are compiled into assembly code running directly on the device. However, the difference seems to be negligible with respect to the advantages deriving from using MIDlet in-lining. Firstly, we must consider that replacing the standard KVM is a very difficult, and often impossible, operation on mobile devices. Indeed, it is a quite complex task for user having no particular technical skills. Furthermore, many manufacturer simply do not allow to modify the KVM of their devices. Instead, being a normal application, the in-liner can be installed on every device with no restriction. Another advantage of using the in-lining method is its higher versatility. In fact, the in-liner can be tuned on the user's necessity by changing the list of API calls to instrument. In this way, different MIDlets can be instrumented only for some operations, while a customised KVM always intercepts all the monitored invocations. Moreover, the instrumentation process can be applied more than one in order to add new operations to monitor, while changing the monitored APIs list of the security enhanced KVM requires a new installation of the virtual machine.

Power saving. Besides the performance overhead, running the PDP generates an increment of battery power consumption as well. We started by comparing the battery consumption of a mobile device running the original web browser to a mobile device running the web browser and, additionally, the PDP as a background process. We expected to see the impact that an additional process has on the battery consumption. Then we also compared the battery consumption loading a local web page on an unmodified system to the battery consumption loading the same local web page on a modified system, i.e. running the in-lined web browser and the PDP. With these set of test we expected to get figures on the power consumption with an active PDP.

To measure the battery power consumption we run the Nokia Energy Profiler[2]. The Energy Profiler monitors various parameters of a mobile device and, in particular, its power consumption. The energy consumption has been measured on the device running the web browser but not performing any security-relevant operation.

The measured data for both scenarios, i.e., without and with PDP, are shown in Figure 3.15. The X axis scale reports the time units. On the Y axis the power consumption in watt (W) is given. Comparing the two graphs we do not notice relevant differences in power consumption (peaks are due to the periodic awakening of system daemons).

In the second test we force to web browser to load a local html document and we observe the power consumption due to the monitoring process applied to this

[2] `http://www.forum.nokia.com/`

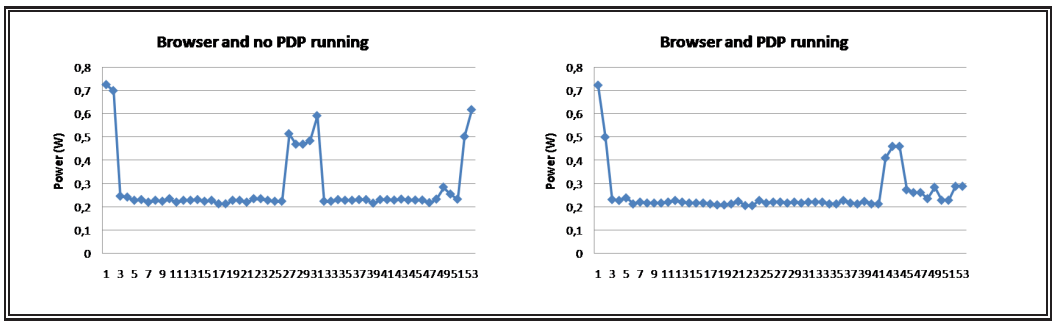


Figure 3.15: Battery power consumption without and with a running PDP.

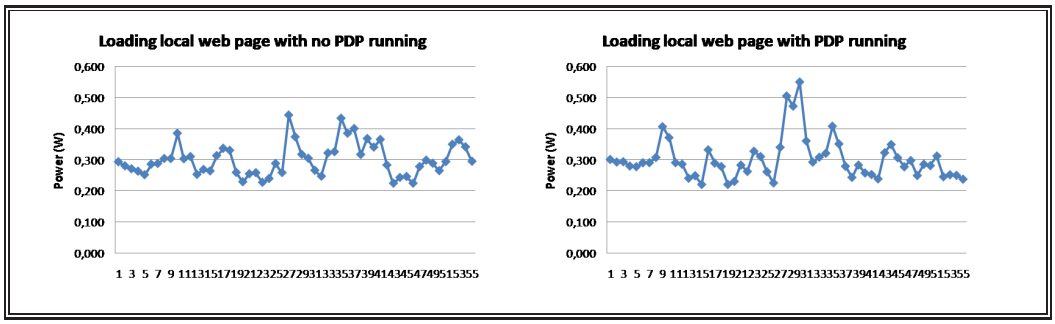


Figure 3.16: Power consumption for loading a local html page.

action. The results are shown in Figure 3.16. Again, even though a small peak is produced by the PDP activation (time 27 – 31), the power consumption profiles are very similar. Hence, we can assert that the monitoring cost in terms of power consumption is totally sustainable by a mobile devices.

3.4 A centralised monitoring architecture for mobile devices

In this section we present a monitoring architecture generalising the model of Section 3.3. We aim at dealing with more realistic assumptions. In particular, we focus on the security of a platform as a whole, rather than on a single aspect or component.

3.4.1 Platform monitoring

In the previous sections of this chapter we focussed on the problem of guaranteeing security properties over the execution of programs. As a matter of fact, almost every security model roughly distinguishes among actors and resources (e.g., programs or

users, and connections or files), and the system, i.e. the platform. Here we re-discuss these assumptions in order to move to a more general characterisation of a system in terms of security-relevant behaviours.

Motivations. In Section 3.3 we introduced runtime monitoring as a technique, used by a system, for verifying that the behaviour of a program, e.g. a Java MIDlet, complies with a security specification. Nevertheless, evaluating the actual security state of a complex device goes beyond the mere control of MIDlets behaviour. As a matter of fact, mobile devices holds many different resources that can be accessed and used in several ways. For instance, mobile phones often execute native binary code as well as dynamically interpreted code, e.g. *Ruby* [163] and *Python* [160]. Security flaws can involve two or more of these components and they all should be considered. Although we can assume that every intermediate language can be instrumented, different execution environment use different abstractions for the access to the actual system resources. Also the level of detail for actions visibility can drastically change. Indeed, observing low level operations, e.g., UDP packages writing and sending, is relatively easier for interpreted instructions then for machine executable code. In general, we have to restrict to the largest set of actually visible events.

A further source of complexity derives from the difficulty of distinguishing between resources and actions. Indeed, several components of a mobile device play a double role. For instance, a GPS receiver is both a resource that needs to be guarded, e.g., for restricting the access to programs, and a source of events, e.g. coordinates changing in time. Moreover, resources and actions are very heterogeneous categories. Some actions could be prevented, e.g., an instrumented Java MIDlet trying to open a connection, while others are unstoppable, e.g., GSM signal blackout. We also need to consider the actions sources, e.g. programs and users. When the set of actions of two different actors partially overlap, the monitoring environment must be able to react in the more appropriate way.

In Section 3.4.3 we apply these considerations to a realistic case study for mobile devices, i.e., a system for parental control.

Apparatuses. One of the main issues when designing a centralised monitoring architecture consists in finding a unique behavioural model suitable for describing all the aspects of the target system. Indeed, finding ad hoc solutions for each security-relevant component would increase the overall complexity. This can make it very difficult to reason about the security of the system in terms of the composition of the behaviours of very diverse objects.

The level of abstraction that we use in our system is based on the key concept of *apparatus*. Basically, an apparatus is a mixed set of software and hardware components mediating the access to some security-relevant entity. More in detail, we identify an apparatus with a homogeneous class of either physical or logical re-

sources, e.g. text messages and GPS receiver, and the interfaces for accessing them, e.g. system APIs. As they are informally defined, an example can better characterise the structure of an apparatus.

Example 3.12 Consider the phone call interface of a generic mobile device. Basically, it can be triggered by several, different actors: a user pressing the "call" button, a program modifying the network settings or a remote agent calling the mobile phone. As it mediates the access to an abstract resource, i.e. the phone network, we can model this part of the device as a system apparatus.

Note that the logical organization of the facilities of a platform in apparatuses is arbitrary. For instance, consider again the phone call apparatus of Example 3.12. Following a similar reasoning we could make a distinction, so defining two different apparatuses, between incoming and outgoing calls. When defining an apparatus we only aim at having a reasonable compromise between a low and an high level view. In fact, the former is usually closer to the actual operations performed by the device but can be far from offering an intelligible representation of the security state. Instead, the latter can be much more descriptive but could neglect some relevant behaviours.

Interestingly enough, we do not even require apparatuses to have a pairwise disjoint scope, i.e. not influencing each other or referring to shared resources. For instance, it seems reasonable to use two different apparatuses for J2ME APIs (see Section 3.3) and the messages interface, e.g. text and multimedia messages. Nevertheless, J2ME also includes libraries for the access to the messaging facilities.

For our purposes, an apparatus must only satisfy three requirements. It must be:

1. *unique*, i.e. there can be at most one instance in the system;

2. *non by-passable*, i.e. it is the only (direct or indirect) access point to a set of (possibly abstract) resources;

3. *observable*, i.e. all the operations involving the apparatus can be seen by a proper observer.

3.4.2 Extensible monitoring architecture

The enforcement system uses a centralised security service. This core service is responsible for evaluating the overall security state and decides how to intervene. Moreover, the basic functionalities can be extended with supplementary modules. Hence, we opted for a plugin-based architecture of the core components. Below we present the architecture of our monitoring system. In particular, we describe the modules composing the monitoring environment and their interaction.

Core service. Following a event-driven model, we implemented the security manager as a centralised, system service, namely the *core service*. Roughly, the core service is responsible for two main activities: (i) managing the security policy and (ii) maintaining the communications with the apparatuses.

The first point is obtained by including a PDP (see Section 3.3) in the service. During system boot, the core service loads the security policy from a secure location and initialises the security state, i.e. loads the values stored during the previous session. This information is used to start the PDP that recovers the state saved at the moment of the last shut down of the device.

Communications are a crucial for the correct behaviour of the system. Basically, the security state of the PDP relies on the continuous arrival of information from the apparatuses. Indeed, a malicious agent can try to alter the security state by modifying, deleting or injecting packets. For this reason we included a communication interface that is responsible for authenticating the communications. This process is obtained by signing the communications and checking their sequence.

The core service automatically starts when the device is turned on and cannot be deactivated neither manually nor by other programs. Actually, the service works as a background process, but is always visible. This means that the user can check its presence in the of the active programs, i.e., using the system task manager, and can put it in foreground, e.g., for reading the current security state. In this way we guarantee that this software cannot be misused for inappropriate purposes. Indeed, an invisible program could be adapted for illegal usages, e.g., remote spying and activity control.

Security modules. A security module is a software package that extends the basic functionality of the control system. Each module must be included in the already working system. The inclusion process is responsible for:

- *Actions/reactions registration.* The definitions of the actions and reactions that the module introduces in the system are appended to the existing alphabet.

- *PEPs deployment.* The module places the PEP that will generate the security actions and interpret the received reactions.

- *Attributes declaration.* The attribute manager is extended with the code for querying the values of the attributes that are relevant for the new security module.

- *Default rules insertion.* The current security policy is extended with the default rules for handling the security actions.

After the installation of a module, the security policies can include rules referring to the freshly registered actions and reactions. Actions and reactions are written

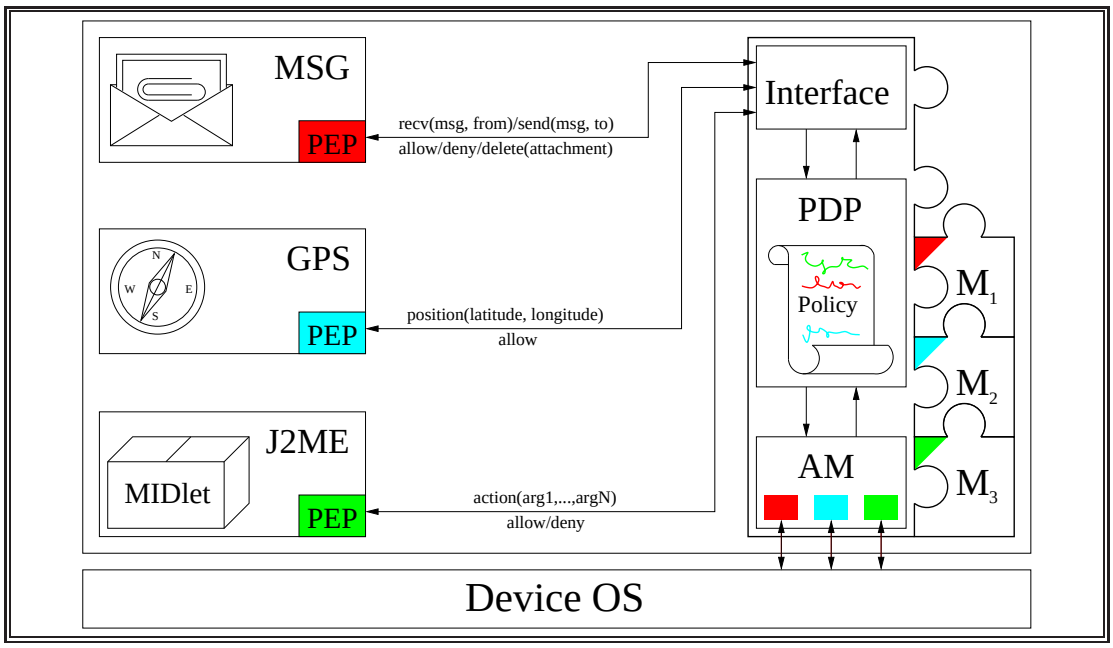


Figure 3.17: A schematic representation of the system managing three modules.

inside private XML files. Roughly, they contain a list of declarations of actions (reactions). Each declaration consists of a symbolic name, a module identifier, an action identifier, a list of typed parameters.

The PEPs are deployed and activated. Each PEP installed by a module must fire actions and handle reactions of the same type of those declared by the corresponding module. We also assume that every PEP can accept two special actions, namely *allow* and *deny*. The messages, i.e. actions and reactions, with the core service pass through a communication interface. In general, each module can deploy a number of PEPs that depends on the features of the monitored apparatus. For instance, monitoring the GPS data can be done through a single manager invoking the proper system APIs. Instead, a large number of PEPs could be required for the control of the Java MIDlets (see Section 3.3).

The attributes are registered by loading in the core service a new object implementing a two-methods interface. Basically, each attribute must simply implement a pair of functions *get* and *set*. Often, only one of the two functions is actually implemented. For instance, we can imagine an attribute mediating the access to the charge level of the device battery. Clearly, the system can read the value but it cannot be modified. The dynamic access to attributed is delegated to a specific system component, namely the *Attribute Manager* (AM).

As our enforcement mechanism follows a *default-deny* approach, a new module must also insert appropriate rules in the system policy. Indeed, if the security policy contains no rules for a certain action, the PDP answers with the *deny* reaction that

is accepted by every PEP. In many cases, this is not the desired default behaviour. Even though it is not always the case, sometimes we can decide to include in the policy a *default-allow* clause for each new action.

The structure of our enforcement environment is depicted in Figure 3.17. Basically, we represented an environment with three monitored apparatuses: the messaging interface (red), the GPS receiver (cyan) and the J2ME platform (green). The modules M_1, M_2 and M_3 extend the core service. The components of the same colour, i.e. PEPs, policy rules and attributes, are created and activated when the corresponding module is deployed. Finally, we used arrows to represent the communications between the PEPs and the interface. Each arrow is labelled with the actions (above) and reactions (below) that the PEP can send and receive.

Security policies. Here we need to model a global security policy, possibly containing many rules. These rules must handle the security state according to the behaviours of heterogeneous entities. ConSpec can offer some advantages. Indeed, a ConSpec policy is a list of security statements. Also, it stores the security state using global variables. In general, reasoning in terms of variables and their values can be easier than using the automata states.

We extended the ConSpec language with a number of features for dealing with the complex operations that we need in a real implementation. The resulting syntax is very rich and articulated. Hence, here we limit the presentation to the most important aspects that we introduced.

A main difference is the structure of the security rules. In our language, a rule is a tuple $\langle \texttt{Action}, \texttt{Guard}, \texttt{ComList}, \texttt{Reaction} \rangle$ where

- `Action` is an action belonging to the accepted alphabet. Each parameter in the action declaration can be renamed in order to use it for the evaluation of the rule;
- `Guard` is a boolean expression. It is obtained combining the classical logical connectives, relations over parameters, variables and constants;
- `ComList` is a sequence of commands. For instance, a command can be the assignment of an expression to a variable;
- `Reaction` is the constructor of a reaction. The reaction is invoked through its unique identifier and its parameters are replaced by actual values obtained during the evaluation of the rule.

Here, a policy is a list of abstract transitions from a security state to another. When the policy is evaluated, all the rules concerning the received action are considered. Among them, we activate those having a guard which is satisfied according to the current state and action parameters. For all the active rules, we execute the corresponding commands. Finally, we synthesize and send back the reaction.

Note that, even though many rules can be active after a single action, we can only generate one reaction. Hence, the system follows a priority relation for deciding the "elected" reaction (e.g., *deny* always wins and *allow* always loses).

The policy is *total* with respect to the actions in the sense that it always generates a reaction. If no rules fit with a received action, the system produces a *deny* reaction. This behaviour complies with the *default: deny* approach. Nevertheless, in many practical cases it turned out that a *default: allow* mechanism would better implement some security requirements. For this reason, we also included special *allow-all* rules that always accept a certain action.

The security policies are stored through signed XML files. These files are crucial for the correct behaviour of the enforcement environment. During the system bootstrap, the policy is verified for integrity and loaded. Moreover, the policy is stored in a protected area of the file system and preserved in multiple copies.

3.4.3 Parental control: a case study

We designed our system in order to have a versatile model that can be easily tuned in for coping with different scenarios. Nevertheless, the system can be tailored to address a specific security domain. Here we show the application of the methods presented so far to the problems of children safety and parental control. The results of this work have been used for the creation of a real protection tool called *iCareMobile* [99].

Minors protection. The problem of protecting the young users from the possible threats deriving from the usage of mobile phones has a relatively long tradition. Since they very first appearance on the market, mobile devices raised issues that attracted the attention of the citizens and institutions. For instance, many discussions are related to the problem of deciding which is the right age for the first mobile phone [4]. In the last years, many public and private associations carried out a control activity on the usage of mobile phones. Some of the most interesting results derived from the study of quantitative as well as qualitative aspects.

The *Pew Research Center* [156] has been monitoring many trends related to the citizens access to the digital technology for the last years. In particular, the *Pew Internet* [155] project was entirely dedicated to this mission. Many periodical reports provided a very descriptive view of several tendencies that emerge from the observation of the American society. In many cases, the consequences of these statistics can be reasonably extended to almost every western country.

From a quantitative point of view, the presence of mobile devices among the minors has been continuously growing. For instance in the US, the percentage of the adults having at least one mobile phone is converging to a stable point, i.e., about the 90%, while the number of young users is still increasing [120]. Also the type of produced traffic and the types of communications are very different. For instance, it turned out that minors mainly use text messaging for their communications.

On the other hand, the usage made by teens is also qualitatively different. In particular, new behaviours have been identified. Among them, *sexting* [121] is getting more and more common. Sexting is the activity of users exchanging sexual contents via text messaging, also called "sexts". The statistics outlined that the 4% of the young users sent and the 15% received sexts.

Another interesting information derives from the behaviour of the minors' parents. As a matter of fact, the average age for the first mobile phone is continuously decreasing [122]. This means that, beyond the actual risks, the parents still consider the mobile phones to be appropriate for their children. Nevertheless, it is also important to remark that when asking teens the reasons why they stopped using a mobile phone, one of the most common answers is "parents took it away" [122]. This remarks the fact that many parents care about the access of their children to this technology.

Contents filtering. The main issue for an actual implementation of a parental control system is to cover the gap between the real, high-level necessities and the internal, low-level model of security. We addressed this point by exploiting the extensional architecture of our system.

In [9] we presented a solution to the problem of detecting the contents that characterise sexts. Basically, we integrated a pattern recognition process, namely *image classification*, in the structure of the module responsible for managing the messaging activities. The process can be applied to the attachments of the incoming and outgoing messages.

In order to perform the automatic classification of images for finding sexual contents we developed a classifier that basically distinguishes between images belonging to a class (images with sexual content in this case) and images that do not belong to the class. We used the *Support Vector Machine* (SVM) [69] technology applied to images described using the MPEG-7 visual descriptors. We decided to use the SVM technology due to its potential flexibility in being adapted to specific applications. Indeed, a number of possible kernels can be used to characterize the space of features and the parameters that can be set for fine tuning. In addition, the decision function of an SVM is conceptually very simple and can be easily implemented also on a mobile device. We refer the interested reader to [9] for a detailed dissertation.

The result of the classification process is a rating value in a continuous range, e.g. $[0, 1]$. In general, the classification of an actual sexual content returns a value close to the higher bound of the range. Hence, we can interpret this value as the level of certainty of the classifier that the image is a pornographic one. Note that the classifier is not responsible for deciding whether to accept or reject a message. Such a decision is still delegated to the PDP. Instead, the result of the classification is communicated to the core service, hosting the PDP, through a proper action. Finally, the PDP evaluates the result with the policy and sends back a reaction that possibly forces the cancellation of the message.

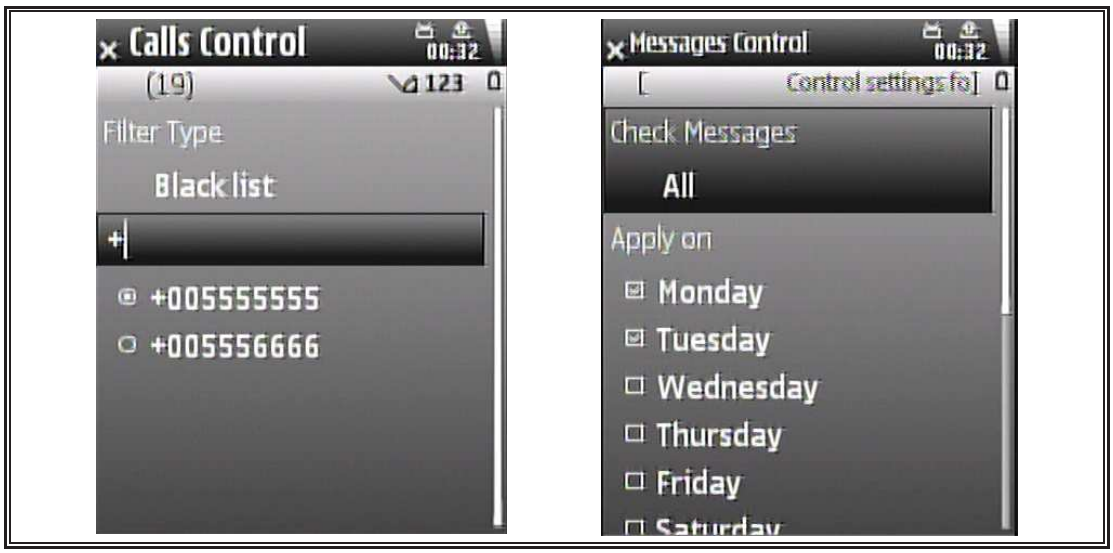


Figure 3.18: Control screens for phone calls (left) and messages (right).

Policy editing. A major issue arising during the design of a usable parental control system is finding a mechanisms that non-skilled users, i.e. the parents, can use to define their security requirements and rules. Indeed, the creation of a security policy is a very complex process requiring a quite rich technical background. Usually, administrators define the security policies for the users of a system. As a matter of fact, this activity requires a good, or even perfect, knowledge of the system, the security mechanisms and the policy language. On the contrary, here we want to expose a set of controls to users that could have a scarce understanding of the mobile device and its components.

In order to overcome this limitation, we designed a graphic user interface for the declaration of the security controls. The idea is to provide the parents with a set of facilities that can be easily compared with the informal recommendations that they usually ask their children to follow. These recommendations can be about the direct usage of the mobile phone. For instance, recommendations like "Do not use games at school" or "call me when you leave from school" are very common. However, some of them can also indirectly involve the device like "Do not get in contact with unknowns (e.g., through phone calls)".

The access to the security controls is protected by a password. When the parent logs in, the system loads the current policy and authenticates to the core service. Then, it is possible to access and modify the security settings. The controls are divided in compounds containing homogeneous elements. Figure 3.18 shows two of them responsible for phone calls and messages.

Other groups of controls are for applications usage, anti-theft settings and position control. When all the settings are submitted, the system saves the current policy

and sends it to the core service through a reliable channel. After the submission, the PDP loads the new policy and starts enforcing the new rules.

3.5 Discussion

This chapter presented our work on the security issues regarding mobile applications. We schematically list the sections of this chapter with their content.

- Section 3.1 proposes an extension of the Java language allowing a code developer for defining and applying local security policies.
- Section 3.2 presents an implementation of the programming model introduced in Section 3.1.
- Section 3.3 describes a tool for centralised security enforcement on resource limited devices.
- Section 3.4 extends the previous model by introducing a plugin-based architecture for monitoring global security properties of mobile devices.

Even though the recent advancements of the mobile technology released some of our working assumptions, e.g., we could argue that modern smart phones are not limited capabilities devices, the security issues deriving from mobile applications are still a main concern. As a matter of fact, in the last few years the diffusion of mobile applications have been rapidly growing. Also, while the number of J2ME-enabled platforms is reducing, other Java-based systems, e.g., Android, are becoming more and more popular.

As the capabilities of the mobile devices increase, we delegate them more responsibility and sensitive tasks. This could make them even more appealing for security attacks. Hence, beyond the continuous evolution of the reference context, all the motivations for the work presented here keep unchanged.

Trust-Driven Secure Composition

To formally analyse a system secure we need a proper representation of both the security policy and of the system itself. However, it is often the case that some parts of a composed system are not available when the security verification takes place. This may happen for several reasons. For instance, the overall system could be the composition of remote platforms communicating through messages.

Often, this kind of composition relies on a description of the components, namely a *contract*. Contracts typically declare the expected behaviour of each component. The reliability of the contracts plays a central role for the correct compositions. When contracts cannot be verified against the actual implementation of the components, we cannot expect to have formal guarantees of secure interactions. Trust-based mechanisms can be used to mitigate the risk of interacting with malicious components.

In this chapter we propose a security paradigm for integrating the trustworthiness evaluation in the composition process. In particular, our model uses the feedback obtained from the behaviour of previous interactions for deciding whether to accept a new composition. To do that, we present a new class of automata, namely *gate automata*, that can be used for defining security and trust policies. On the one hand, these policies drive the monitoring process generating trust feedback cited above. On the other hand, the trust feedbacks generate a continuous tuning of the security constraints in order to increase the controls over the untrusted components and to decrease the security overhead for the trusted ones.

4.1 Security-by-Contract-with-Trust

Here we present *Security-by-Contract-with-Trust* (S×C×T), that is a new paradigm for dealing with security and trust in composite systems. The main novelty of our approach consists in obtaining a fully integrated environment for dealing with both trust and security policies. We achieve this by extending the Security-by-Contract model with special facilities for trust management, trust policies enforcement and contract monitoring.

4.1.1 Security-by-Contract Paradigm

We start by recalling the *Security-by-Contract* (S×C) [81] paradigm in its original formulation. The S×C paradigm provides a full characterisation of the contract-based interaction. The two main categories of items in S×C are: *contracts* and *policies*. A contract is an over-approximation of all the possible execution behaviours of an application. Loosely speaking, a contract contains a description of the relevant features of the application and the relevant interactions with its hosting platform. The contract is released by the developer or vendor that provides it together with the application.

The other cornerstone of the S×C approach is the concept of *policy*, which is usually defined on the hosting platform, that is the execution environment. It may be specified by the owner or by the producer of the platform and consists of a set of admitted application execution behaviours.

The core idea behind the Security-by-Contract approach is depicted in Figure 4.19. When a client receives an application, the system automatically checks the formal correspondence between the program code and contract (Check Evidence). This step is intended to provide a formal proof that the contract effectively denotes the behaviour of the running program. This step can be implemented, for instance, using the *model-carrying code* [170] method. If the result is negative, i.e., the contract is corrupted or incorrect, the program runs under the control of a security enforcement agent (Enforce Policy). Otherwise a matching between the contract and the policy is performed to establish if the contract is also compliant with the security requirements. If it is the case, then the application is executed without overhead (Execute Application), otherwise the policy is enforced again (Enforce Policy). Finally, if the previous checks were positively passed, the application can be executed with no active runtime monitor.

The contract-policy matching function ensures that any security relevant behaviour declared by the contract is also allowed by the policy. This matching can be implemented using different behavioural relation, e.g., language inclusion [79] or simulation relation [95]. The matching function allows the user to check whether the behaviour of the application is compliant with the policy or not, without the need for running the application.

Also the enforcement process has been shown to be feasible using different approaches. For instance, two techniques suitable for interpreted languages have been detailed in the literature and exploited for experiments and tools: interpreter customization (e.g., see [52]) and intermediate language rewriting (e.g., in [70, 66]). Briefly, the first replaces the standard system interpreter (e.g., a JVM) with a modified one dispatching signals to the monitoring agent whenever a program makes a call to (a subset of) the system APIs. The second instruments the sequence of interpreted instructions (e.g., the bytecode) with invocations to the security policy monitor making the program send security signals at run-time. Both the approaches typically use an external component, namely a *Policy Decision Point* (PDP, see Sec-

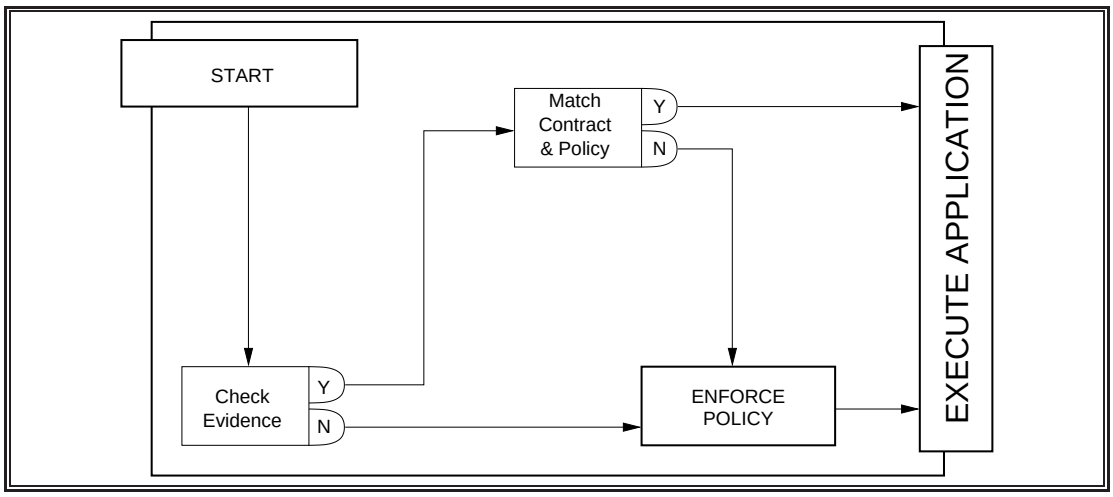


Figure 4.19: The Security-by-Contract process.

tion 3.3), holding the set of rules that compose the security policy. Moreover the PDP reads the current device state (battery consumption, link strength, available credit) through dedicated internal components. When the PDP receives a request for an action violating the security policy, it answers denying the necessary permission. Then, the system reacts by throwing an exception.

Note that the notion of trust was not integrated in the Security-by-Contract approach. Basically, the user can trust or not the application's provider according to a software certification delivered by some certification authorities.

4.1.2 Extending S×C with Trust

As mentioned above, a crucial point of the S×C model is the verification of the relation that exists between the application and its contract. Nevertheless, the approach of many real-world systems to the security of the mobile applications is much more naive. Very often, the mobile code is run if its source is somehow trusted. This means that we can only reject or accept the signature of the application provider. For this reason, it is becoming more and more common among the platform manufacturers to have official applications stores (e.g., see [13, 145, 14]).

Here we propose an extension of the existing architecture by adding a component for the contract monitoring. Checking whether the execution of an application adheres to declared contract we can modify the level of trust of the application provider.

Driving trust management with contract monitoring offers several advantages. Many of them derive from the automatic management of the decisions about the trustworthiness of the providers. Basically, in order to have a precise trust measure, a behavioural feedback is needed. Indeed, the trust weight associated to a certain

provider should change according to whether its applications behave correctly or not. In some cases, the trust value modification is triggered by a program trying to execute some forbidden action, i.e. violating a security policy. However, policy violations are not always caused by a real security attack. As a matter of fact, in case of customised security policies, we can not expect that every provider is aware about the requirements of each customer. Hence, a policy violation could simply be the result of a mismatch between the provider's specification and the platform restrictions.

On the other hand, a precise trust value has some clear advantage for the platform. A first benefit derives from the possibility of avoiding security enforcement. Two important issues about policy enforcement are execution overhead and semantics interference. In other words, when an enforcement mechanisms follow the execution of a program, it slightly decreases the standard performances of the whole system. Furthermore, enforcing a policy may cause a modification of the original behaviour of programs. If a program comes from a trusted source, the user could decide to run it free from any control.

A further important issue is policy specification. Users applying customised security properties on their devices must specify all the acceptable behaviours of running applications. Needless to say, this can be a cumbersome task especially for users having no technical skills. In fact, many users do not specify any security restriction on the applications they trust, e.g. utilities provided by the device's manufacturer. Hence, allowing trusted application to run according to their contracts may also reduce the complexity of specifying security policies.

Our strategy takes place in two phases: at deploy-time by setting the monitoring state and at run-time by applying the contract monitoring procedure for adjusting the provider trust level.

Deployment Architecture

The S×C paradigm does not require the software provider to be a trusted entity and simply relies on the correctness of local, internal components (i.e. Check Evidence and Contract-Policy Matching). Here, we deploy a framework for quantitative trust management. In this way, it is possible to dynamically update the levels of trust. The Updating is done according to the adherence between the real execution of the application and its contract. Indeed, we extend the existing architecture by adding a *contract monitoring* that checks the compliance between the application and its contract. We replace the contract verification (Check Evidence) with a simple check on the level of trust of the provider (Trusted Provider). If the provider is untrusted then the policy is enforced, otherwise the contract-policy matching is performed. Hence the S×C×T workflow results as in Figure 4.20 and Figure 4.21. It consists of the following steps:

- **Step 1-*Trust Assessment*:** Each downloaded mobile application is associated with a given recommendation rate, which allows the trust module of the

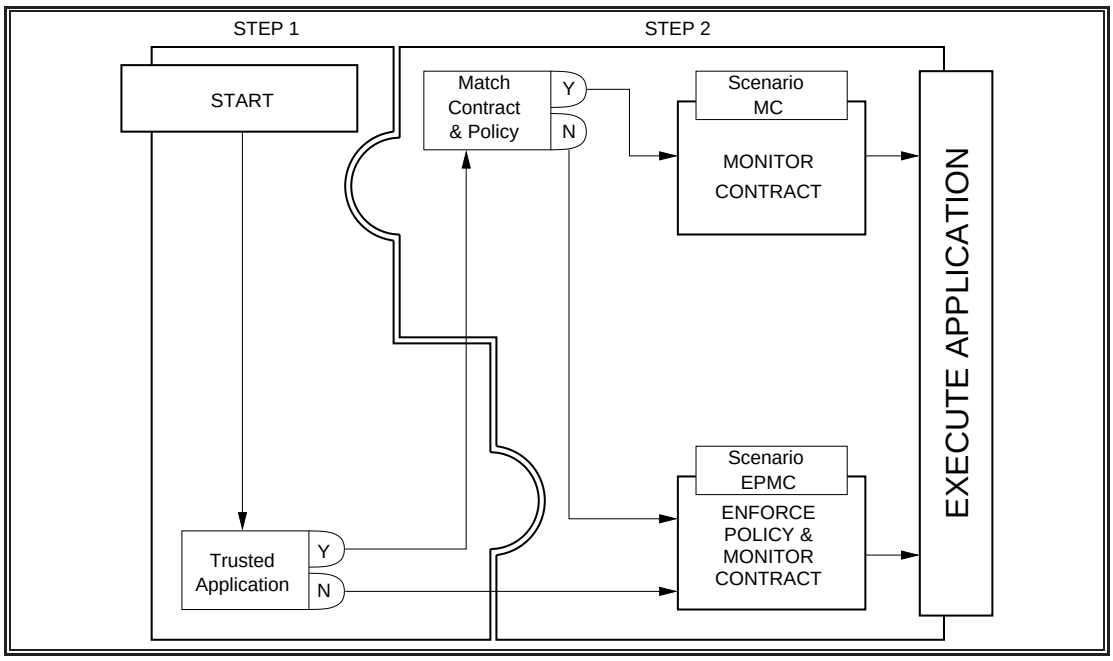


Figure 4.20: The extended Security-by-Contract application workflow.

user device to decide if the application can be considered as trusted or not (see section 4.1.3).

- **Step 2-*Contract Driven Deployment*:** According to the trust measure, the security module decides to simply monitor the contract or also enforce the policy going into one on the scenarios described in Step 3.
- **Step 3-*Contract Monitoring vs Policy Enforcement*:** Depending on the chosen scenario the security module is in charge for monitoring either the policy or the contract and saving the execution traces (logs).
- **Step 4-*Trust Feedback Inference*:** Finally, the trust module parses the produced logs and infers a trust feedback (see section 4.1.3).

Description of the Scenarios

We outline how trust measures assigned to security assertions can be adjusted as a result of a contract monitoring strategy. Indeed, trust measures associated with the provider concern on the contract goodness mainly. Updated trust measures will influence on future interactions with an application and contract providers. In other words, our system highly penalizes the provider more when the contract does not specify the application's behaviour correctly. Instead, when the application violates the security policy, the system slightly penalizes it.

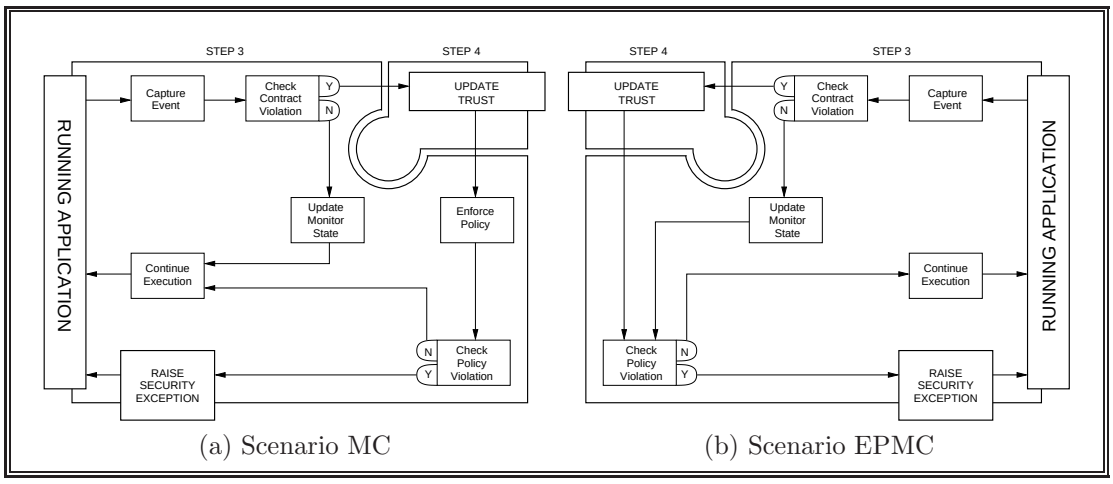


Figure 4.21: The contract monitoring configurations.

We presented in Section 3.3 a proposal for a monitoring infrastructure. It consists of a PDP that holds the actual security state and is responsible for accepting or rejecting new actions. *Policy Enforcement Points* (PEPs) are both in charge of intercepting actions to be dispatched to the PDP and preventing the execution of illegal operations.

Starting from this model, we extend it by making the PDP also responsible for the contract monitoring operations and for the trust vector updating. According to [52, 66], we assume that both contracts and policies are specified through the same formalism. Hence, the policy enforcement configuration of the PDP keeps unchanged. The PDP must load application contracts as well as security policies dynamically. Moreover, it must be able to run under the two different execution scenarios of Figure 4.20. The two possible configurations are described below.

EPMC Scenario. Both the policy enforcement and the contract monitoring are active. During the execution, contract violations are checked to update the trust levels. Moreover, the policy enforcement guarantees that the application does not violate the security policy. In particular, the contract monitoring receives event signals from the executing code and keeps trace of the execution trace. When a signal arrives, its consistency with respect to the monitored contract is checked. If the contract is respected then the internal monitoring state is updated and the operation is allowed, and a good behaviour is logged (i.e., contract respected). Otherwise, if a violation attempt happens, a security error occurs and a violation feedback is logged for the trust module. The policy enforcer is in charge for following the execution of the application. Whenever the program attempts to violate the security policy, the enforcement mechanism halts the execution. This guarantees that the security policy is always satisfied.

MC Scenario. The contract monitoring is performed. It works according to the following strategy: the contract monitoring receives event signals from the executing code. The execution trace is kept in memory. When a signal arrives, its consistency with respect to the monitored contract is checked. As in the previous case, if the contract is respected then the internal monitoring state is updated and the operation is allowed, and a good behaviour is logged. Otherwise, if a violation attempt happens, a security error and a bad trust feedback occur. After a violation, the system switches from the contract monitoring to the policy enforcement configuration in order to guarantee that the security policy is satisfied.

The whole behaviour of the two configurations is depicted in Figure 4.21.

4.1.3 Trust management

Trust management techniques are used in systems where some level of uncertainty exists upon the components and their behaviour. In many practical cases, a client has no formal evidences guaranteeing the correct behaviour of a software or a service before the actual execution. Nevertheless, in real life many transactions and collaborations take place without formal assurances. In fact, we can argue that most of the interactions are based on clients' expectations and providers' promises. Hence, users rely on *trustworthy applications* for many of their daily activities. On the one hand, this behaviour pragmatically simplifies many security issues increasing the usability of several systems. On the other hand, attacks trying to exploit the methodical evaluation errors due to the psychology of the human beings are even too simple, e.g. *phishing*.

In the last years, much work aimed at providing a rigorous description of the conscious and unconscious evaluations about trustworthiness. As a consequence, several models of trust have been proposed (see [152] for a survey). Due to the nature of our model, i.e., a system continuously adjusting the trust values, we decided to use a quantitative representation of trust.

In this way, we associate each application provider to a certain trust weight ranging in $[0, 1]$. The *trust manager* (TM) is the component responsible for handling these values. Basically, it consists of a mapping between the unique identifier of a provider and a trust weight. Also, it implements the basic operations for *rewarding* and *penalising*. Intuitively, a reward (penalty) is a trust feedback causing the increasing (decreasing) of the trust level of a provider. In general, there can be more that just one type of reward (penalty) in order to model different feedback types, e.g., feedback related to the provider community or identity.

In [65] we proposed a trust manager using two types of feedback, i.e. *high* and *low*, for rewarding and penalising. The trust manager is implemented as a centralised component of an applications marketplace where providers register they programs and users retrieve software for their platforms. Each feedback from authenticated

Trust \ Security	Respect the contract		Violate the contract	
	MC	EPMC	MC	EPMC
Trusted	High reward	Low reward	High penalty	
Untrusted	——	Low reward	——	Low penalty

Table 4.5: The trust feedback generated by an application.

users changes the level of trust of the application provider within the marketplace. The trust level is used as a recommendation factor for the new customers of the provider. A user bases the decision about the trustworthiness of the provider on the recommendation of the marketplace. Table 4.5 lists the trust feedback types that users can send according to the model of [65].

Briefly, untrusted applications, i.e. coming from an untrusted provider, can only run under the EPMC configuration. We decided that they receive a low feedback for both respecting or violating the contract. Instead, for trusted applications we apply a different mechanism. We always punish the contract violation severely and we reward the contract compliance according to the execution configuration, i.e. high for MC and low for EPMC.

Note that this table is purely arbitrary. It encodes the procedure for managing the trust feedback. Changing the number of feedback and the cells content we can obtain different behaviours.

4.2 Introducing Gate Automata

In this section we introduce *gate automata* and their properties. Moreover, we shall provide the reader with several examples showing how gate automata can be suitably used for specifying security and trust policies. These policies are suitable for integrating security and trust in the S×C×T runtime environment. Indeed, a gate automaton represents a security policy that, as a side effect, can trigger the S×C×T trust manager. On the other hand, the trust manager is responsible for deciding the security domain of an application, i.e., which are the security policies to be applied on it.

Also note that this model offers a precise characterisation of the relation between trust and security in the S×C×T. As a matter of fact, we create a reciprocal dependency between the two concepts. In particular, trust increasing (decreasing) may cause a decreasing (increasing) of the number of the active security policies. Similarly, more security policies may produce more feedback for the trust manager.

In this way, we generalise the S×C×T runtime by relaxing the limitation of having two configurations, i.e., MC and EPMC. Through gate automata we achieve a number of configurations, i.e., approximatively one for each policy, that are dynam-

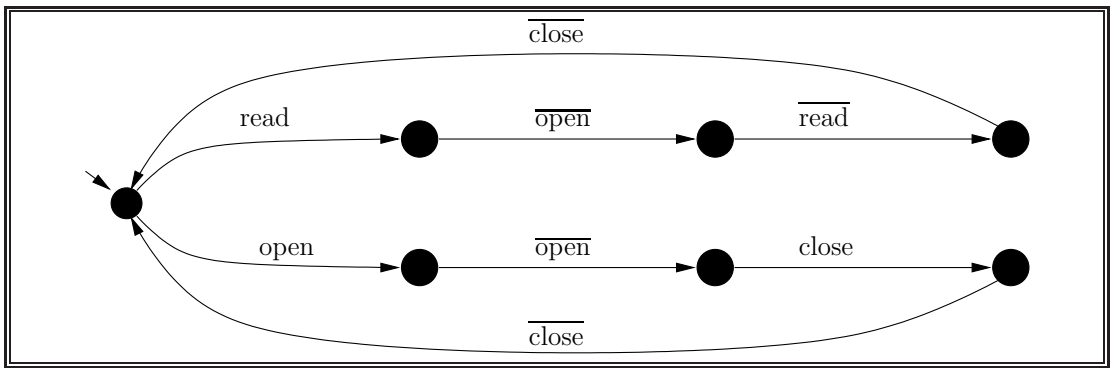


Figure 4.22: A gate automaton for file access.

ically updated at runtime.

4.2.1 Gate automata

Intuitively, a gate automaton denotes a security policy in the form of a reacting agent. When it receives a security-relevant action, it starts a reaction procedure involving an arbitrary number of steps. Each step consists of either a visible, output action or an internal, trust feedback. An external observer can only see the output actions. According to the S×C×T paradigm, the interpretation of the trust feedback is delegated to a centralised trust manager that collects them.

Before giving the definition of usage automaton, we propose the following example to clarify the structure and behaviour of the automata.

Example 4.13 Imagine a *file access* policy φ_{FA} saying "never read a file if it is not open". The gate automaton of Figure 4.22 represents φ_{FA}.

The automata has two possible paths starting from the initial (leftmost) state. In the initial state, the automaton waits for an input action. All the actions which label no outgoing transitions are neglected, i.e., the automaton does not change them. When an action "read" is performed, the automaton takes the top path. Instead, if an action "open" arrives, the automaton goes through the bottom path. In the first case, the reaction to the input is a sequence of three output actions, i.e. "$\overline{\text{open}}$", "$\overline{\text{read}}$" and "$\overline{\text{close}}$", returning to the initial state. In other words, whenever a program tries to read a closed file, the automaton wraps this action between a "$\overline{\text{open}}$" and a "$\overline{\text{close}}$". The second path, propagates the action open with no changes, i.e., reads and writes it, and moves to a new state. In this state there are no transition labelled with "read" and "open". Hence, they are allowed. Instead, when a "close" arrives, the automaton makes it pass and returns to the initial state.

We now give the formal definition of gate automata.

Definition 4.1 A gate automaton $\mathcal{G}$ is a 4-tuple $\langle V, \imath, A, T \rangle$ where

- V is a finite set of states;
- $\imath \in V$ is the initial state;
- A is a set of actions;
- $\mathbb{L}(A) = A \cup \bar{A} \cup \{\blacktriangle, \blacktriangledown\}$ is a set of labels such that:
  1. $\bar{A} = \{\bar{\alpha} \mid \alpha \in A\}$ is the set of output labels
  2. $\blacktriangle, \blacktriangledown \notin A \cup \bar{A}$ are the trust labels
- $T \subseteq V \times \mathbb{L}(A) \times V$ is a set of labelled transitions such that:
  1. $(v, a, u) \in T \wedge (v, b, w) \in T \wedge a = b \iff u = w$
  2. $\forall (v, a, u) \in T . a \in \bar{A} \cup \{\blacktriangle, \blacktriangledown\} \implies \nexists\, b, w . b \neq a \wedge (v, b, w) \in T$

A gate automaton slightly differs from a deterministic, finite state automaton (DFA). It has a finite set of states (V, $\imath$ being the initial one), an alphabet of accepted symbols (A) and a set of transitions (T). The transitions of the automata are labelled with either input or output actions. Slightly abusing the notation, we use "actions" in place of *labels* when there is no ambiguity. We write $\alpha \in A$ to denote an input action, $\bar{\alpha} \in \bar{A}$ for an output one and a, b for generic elements of $\mathbb{L}(A)$. Gate automata can also perform two special operations, i.e., $\blacktriangle$ and $\blacktriangledown$, that denote the positive and negative trust feedback. The two supplementary requirements on T force gate automata to be deterministic (1) and limit the outgoing transitions labelled with output or trust actions to be at most one per state (2). In particular, the second restriction characterises the model of reaction. In other words, if a state is the source of some reaction transition, no other transitions are allowed. Where it improves the readability, we use $v \xrightarrow{a} w$ in place of $(v, a, w) \in T$ and $v \not\xrightarrow{a}$ for $\nexists w \,.\, (v, a, w) \in T$.

Producing trust feedback is one of the distinguishing features of gate automata. We propose the following examples to clarify their behaviour.

Example 4.14 Imagine a *Chinese Wall* policy φ_{CW} saying "never send network messages while accessing the file system".

We implement this policy though the gate automaton of Figure 4.23. The target can take two directions. Indeed, it is allowed to either open a file or send data. Each of the previous actions leads to a new state. This state allows all the other actions, but the complement of the previously executed one, i.e. "open" after "send" and "send" after "open". If the second, the relevant action is performed, the automaton does not propagate it, fires a negative trust feedback and moves to a pit, looping state where everything is permitted but the freshly cancelled action.

Note that, from a security point of view, we could implement the same policy in several ways. For instance, we may anticipate the pit states in the path and remove the trust feedback. Nevertheless, the two automata represent two different trust policy.

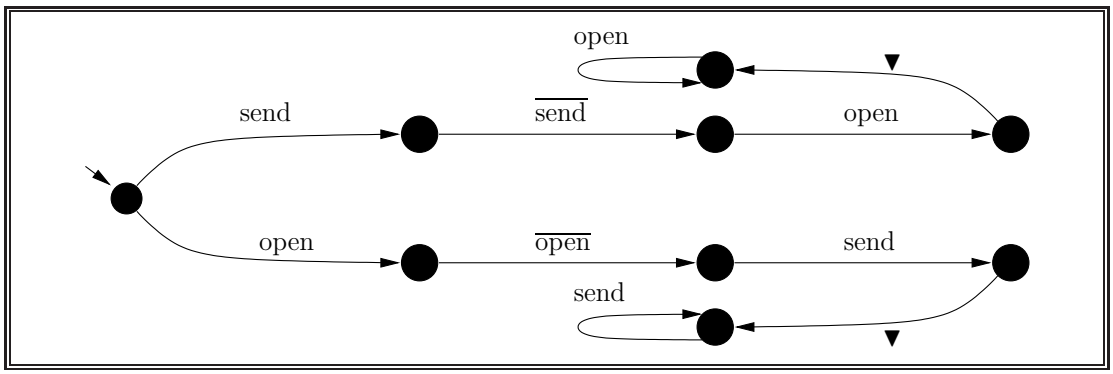


Figure 4.23: A gate automaton for the *Chinese Wall* policy.

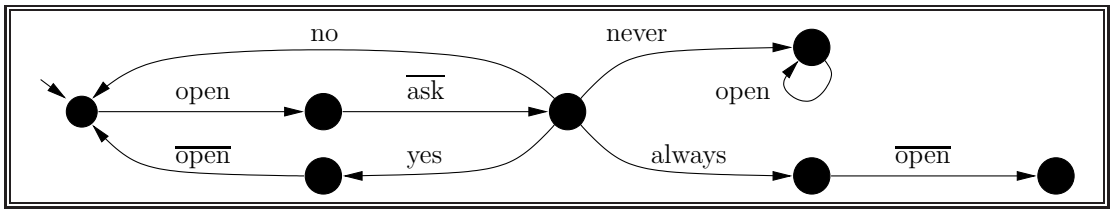


Figure 4.24: A gate automaton for the *ask user* policy.

Example 4.15 The security of mobile devices is based on software certification (see Section 3.3). Basically, an application is signed with a certificate provided by some trusted entity, i.e., a certification authority. At install time, the signature is verified and, if it is valid, the application receives all the required access privileges. If the signature is corrupted or absent, the application has no access rights to security critical operations and every decision is delegated, time by time, to the user. Users allow or deny permissions to each single operation. If the user considers a program to be harmless, i.e., he trusts the application, he can decide to always permit the action. Symmetrically, if the application is not trusted, the user can decide to never allow the access.

The gate automaton of Figure 4.24 implements the policy that asks the user to decide whether to permit once ("yes"), always permit ("always"), deny once ("no") or never permit ("never") the "open" action.

Briefly, when the automaton receives an "open" action, it blocks it and requests the authorization of the user ("$\overline{\text{ask}}$"). The user decides among four answers, each of them causing a different reaction.

- "yes". The automaton fires the "$\overline{\text{open}}$" action and returns to the initial state (waiting for further requests).
- "no". The automaton returns to the initial state and the "open" action is cancelled.

- "always". The automaton generates the "$\overline{\text{open}}$" action and moves to a state where all the actions are permitted.
- "never". The automaton reaches a state where all the "open" actions are definitively blocked (self loop).

4.2.2 Automata semantics

In this section we introduce the relation between gate automata and *interface automata* [71]. In this sense, a gate automaton denotes a corresponding interface automaton, i.e., its instantiation. Therefore the semantics of a gate automaton is given through that of the corresponding interface automaton. The interface automaton obtained in this way reads an execution trace, i.e., its input, and writes a modified trace, i.e., its output, which is compliant with the policy specification. Here, we briefly recall the definition of interface automaton given in [71] (see Section 2.3 for further detail).

Definition 4.2 An interface automaton $P = \langle V_P, V_P^{init}, A_P^{\mathcal{I}}, A_P^{\mathcal{O}}, A_P^{\mathcal{H}}, T_P \rangle$ consists of the following elements:

- V_P is a set of states.
- $V_P^{init} \subseteq V_P$ is a set of initial states such that $\sharp V_P^{init} \leqslant 1$.
- $A_P^{\mathcal{I}}$, $A_P^{\mathcal{O}}$ and $A_P^{\mathcal{H}}$ are mutually disjoint sets of input, output and internal actions.
- $T_P \subseteq V_P \times (A_P^{\mathcal{I}} \cup A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}}) \times V_P$ is a set of transitions.

The main advantage deriving from using interface automata is that they can be easily composed in several ways. In particular, they can be aligned using their interfaces for obtaining a sequential composition. Each automaton receives invocations from its predecessor, changes its internal state, possibly executing internal actions, and invokes the methods of its successor. Using this approach we can implement a chain of policies as discussed in the next section.

A gate automaton can be instantiated to a corresponding interface automaton through a simple transformation. Hence, we use interface automata for giving an operational semantics to the security policies defined through our gate automata. The formal definition on the instantiation of a gate automaton is provided in Table 4.6.

Roughly, the instantiation works as follows. The set of states V_P is obtained by adding a set V_{id} of special states for self loop transitions to the set V of states of the gate automaton. For each state $v \in V$ being the source of some input transition and for each input action α that labelling no transitions from v, we have a corresponding state, namely $v_{id}^{\alpha} \in V_{id}$. The input and output actions of the interface automaton are obtained by pairing the instantiation index k (input) and its successor (output)

An instantiation of a gate automaton $\mathcal{G} = \langle V, \imath, A, T\rangle$ over a index $k \in \mathbb{N}$, denoted by $\mathbf{G}_k$, is an interface automaton $P = \langle V_P, \{\imath\}, A_k^{\mathcal{I}}, A_{k+1}^{\mathcal{O}}, \{\blacktriangle, \blacktriangledown\}, T_P\rangle$ where:

- $V_P = V \cup V_{id}$ is the set of states with

$$V_{id} = \{v_{id}^{\alpha} : v \in V \ \wedge \ \exists \beta \in A, u \in V. v \xrightarrow{\beta} u \ \wedge \ v \not\xrightarrow{\alpha}\};$$

- $A_k^{\mathcal{I}} = \{\langle \alpha, k\rangle : \alpha \in A\}$ is the input alphabet;
- $A_{k+1}^{\mathcal{O}} = \{\langle \alpha, k+1\rangle : \alpha \in A\}$ is the output alphabet;
- T_P is a set of transitions defined as:

$$\begin{array}{rcll} T_P & = & \{(v, \langle \alpha, k\rangle, w) : (v, \alpha, w) \in T\} & \text{(input)} \\ & \cup & \{(v, \langle \alpha, k+1\rangle, w) : (v, \bar{\alpha}, w) \in T\} & \text{(output)} \\ & \cup & \{(v, \blacklozenge, w) : (v, \blacklozenge, w) \in T\} & \text{(internal)} \\ & \cup & \{(v, \langle \alpha, k\rangle, v_{id}^{\alpha}) : v_{id}^{\alpha} \in V_{id}\} & \text{(loop input)} \\ & \cup & \{(v_{id}^{\alpha}, \langle \alpha, k+1\rangle, v) : v_{id}^{\alpha} \in V_{id}\} & \text{(loop output)} \end{array}$$

where $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$

Table 4.6: Gate automata instantiation.

with the input and output actions of the gate automaton. The instantiation index k is used for distinguishing the input from the output actions and also for allowing the composition among the automata as discussed in Section 4.3.2. The labels for trust feedback are used as internal actions. The input, output and internal actions are used to label the transitions the interface automaton. In particular, for each transition of the gate automaton, its instantiation has a corresponding one. Moreover, we add input (output) transitions to (from) the elements of V_{id}. These pairs of transitions define a two-steps path passing through the states of V_{id} and represent allowed actions.

Example 4.16 Consider the gate automaton of Example 4.13. We instantiate it with index k and we obtain the interface automaton of Figure 4.25.

For the sake of simplicity, we use $\alpha?$ and $\alpha!$ in place of $\langle \alpha, k\rangle$ and $\langle \alpha, k+1\rangle$ when there is no ambiguity. Self loops labelled with an action α are a compact notation for the pair of transitions $(v, \alpha?, v_{id}^{\alpha})$ and $(v_{id}^{\alpha}, \alpha!, v)$, where v_{id}^{α} is the small black state in the loop. We use this notation in order to have a more readable representation of the automata.

For what concerns the semantics of an instantiation $\mathbf{G}_k$ of a gate automaton $\mathcal{G}$, we define it in terms of *reaction sequences*. Intuitively, a reaction sequence is a trace of output and internal actions fired by an interface automaton after reading

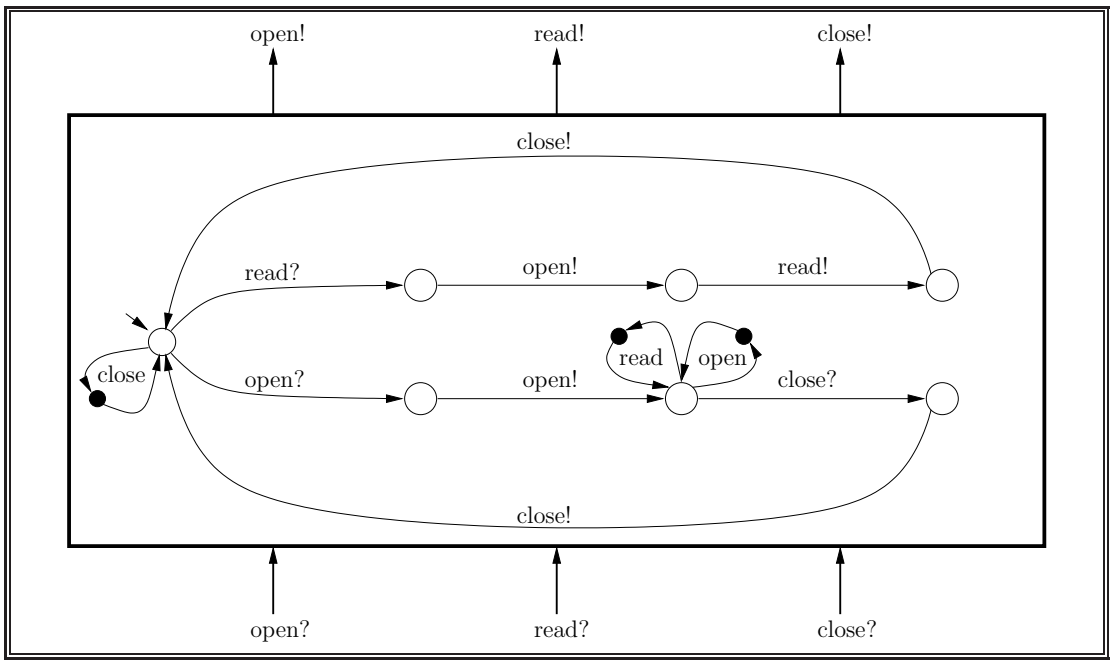


Figure 4.25: The instantiation of the file access gate automaton.

one input symbol. We start by extending the definition of *execution fragment* in the following way.

Definition 4.3 An *execution fragment* of an interface automaton P is a possibly infinite, alternating sequence of states and actions $v_0, \alpha_0, v_1, \alpha_1, \ldots$ such that $(v_i, \alpha_i, v_{i+1}) \in T_P$.

Definition 4.4 Given an interface automaton $P = \langle V_P, V_P^{init}, A_P^{\mathcal{I}}, A_P^{\mathcal{O}}, A_P^{\mathcal{H}}, T_P \rangle$, an action $\alpha \in A_P^{\mathcal{I}}$ and a state $v \in V_P$, a *reaction sequence* to α in v is a possibly infinite trace of actions $\sigma = \alpha_0, \alpha_1, \ldots$ such that

- $\alpha_i \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}}$,
- $\exists v, v_0, v_1, \ldots \in V_P$ such that $v, \alpha, v_0, \alpha_0, v_1, \alpha_1, \ldots$ is an execution fragment of P and
- if σ has finite length n then $\forall \beta \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}} \,.\, v_n \stackrel{\beta}{\not\rightarrow}$.

We say that α is an *activator* of σ in v and denote it with $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} v_n$ if σ is finite or $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} \uparrow$ otherwise.

Example 4.17 Consider the interface automaton of Figure 4.25. Starting from the initial state the automaton takes the top path if it receives an input action read?. The path consists of three transitions (labelled with open!, read! and close!

respectively) and returns to the initial state. Hence, calling $\imath$ the initial state, we say that read? activates the reaction open!read!close! in $\imath$.

4.2.3 Trace validity

In this section we provide a formal definition of compliance of a trace with respect to a gate automaton. Intuitively, we can imagine that a sequence of actions is allowed by a gate automaton if, passing it as the input of the (instantiation of the) automaton, the output is the unchanged sequence. Below we formally define this notion in terms of reactions sequences.

As the input and output actions of the instantiations of gate automata are paired with the instantiation index, we use an operator, i.e., the function f_{out} to remove them and obtain standard sequences of actions. This function also removes the trust feedbacks which are not visible from outside the automaton. The function f_{out} is recursively defined as follows.

$$f_{out}(\langle \alpha, k\rangle) = \alpha \qquad f_{out}(\langle \blacklozenge, k\rangle) = \cdot \qquad f_{out}(\sigma\sigma') = f_{out}(\sigma) f_{out}(\sigma')$$

being $\cdot$ the empty trace and $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$.

We can now define the notion of weak compliance for the gate automata.

Definition 4.5 Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T\rangle$ we say that σ is *weakly compliant* with $\mathcal{G}$, in symbols $\sigma \vdash \mathcal{G}$, if and only if for any instantiation $\mathbf{G}_k$ of $\mathcal{G}$ we have

$$\imath \xRightarrow[\langle \alpha_1, k\rangle]{\sigma_1^{k+1}} v_1 \ldots \xRightarrow[\langle \alpha_n, k\rangle]{\sigma_n^{k+1}} v_n$$

such that $\sigma_i^{k+1} = \langle \beta_{i,1}, k+1\rangle \ldots \langle \beta_{i,m_i}, k+1\rangle$ and

$$f_{out}(\sigma_1^{k+1} \ldots \sigma_n^{k+1}) = \sigma$$

Beyond the technical definition, we can see the weak compliance as the dual of *transparency*. That is, a trace weakly complies with a gate automaton if and only if an external observer cannot understand whether the trace has been processed by (the instantiation of) the automaton or not. The following example can help in clarifying this aspect.

Example 4.18 Imagine a policy φ_{RC} saying "all files must be closed after reading". We represent this policy through the 4-states gate automaton $\mathcal{G}$ depicted below.

The automaton $\mathcal{G}$ allows a "read" action and immediately enqueues a "close" reaching the rightmost state w. From this state two branches are possible. If the next action is a "close", the automaton cancels it and returns to the initial state $\imath$. Instead, if it receives a "read" action, it loops on the second state u and repeats the first behaviour.

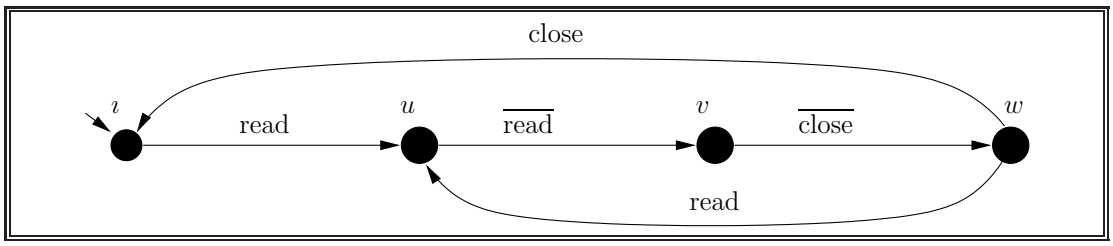


Figure 4.26: A *file closing* policy.

Consider now the trace $\sigma = \text{read}, \text{close}$. It is easy to verify that interface automaton obtained instantiating the gate automaton for φ_{RC} reacts to σ in the following way:

$$\iota \overset{\substack{\text{read!,close!} \\ \text{read?}}}{\Longrightarrow} w \overset{\substack{\cdot \\ \text{close?}}}{\Longrightarrow} \iota$$

where we used the notation of Example 4.16 for input and output actions.

Since $f_{out}(\text{read!}, \text{close!}) = \sigma$, the trace is weakly compliant with $\mathcal{G}$, i.e. $\sigma \vdash \mathcal{G}$.

Clearly, weak compliance does not correspond to a full transparency. Indeed, the transitions of the automaton can introduce and delete actions in such a way that a trace is kept unchanged as a whole, but its prefixes are modified. For instance this can happen when the automaton anticipates an action, e.g. close in the previous example, or postpones it.

For characterising sequences that are not modified at all by a gate automaton we use the notion of *strong compliance*.

Definition 4.6 Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ we say that σ is *strongly compliant* with $\mathcal{G}$, in symbols $\sigma \models \mathcal{G}$, if and only if for any prefix σ' of σ holds that $\sigma' \vdash \mathcal{G}$.

Example 4.19 Consider again the gate automaton $\mathcal{G}$ of Example 4.18. We already discussed the weak compliance of the trace $\sigma = \text{read}, \text{close}$ with respect to $\mathcal{G}$. However, we also observed that

$$\iota \overset{\substack{\text{read!,close!} \\ \text{read?}}}{\Longrightarrow} w$$

and $f_{out}(\text{read!}, \text{close!}) \neq \text{read}$. Hence, σ is not strongly compliant with $\mathcal{G}$.

Comparing gate automata with edit automata Starting from the definition of security automaton given by Schneider in [168], Ligatti *et al.* [123] have defined a new category of deterministic security automata, namely *edit automata*. The importance of the edit automata resides in the fact that they identify a very general class of properties, namely *edit properties*, suitable for runtime enforcement. Here we

show in a constructive way that the properties expressed through gate automata are a subset of the edit properties. Translating a specification into an edit automaton has some advantages. For instance, one can exploit existing tools based on edit automata, e.g., *Polymer* [31].

An edit automaton is defined as $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$, where $\delta : A \times \mathcal{Q} \to \mathcal{Q}$ is the transition function, $\gamma : A \times \mathcal{Q} \to A \times \mathcal{Q}$ specifies the insertion of an action into the program actions sequence and $\omega : A \times \mathcal{Q} \to \{-, +\}$ indicates whether or not the action in question must be suppressed (-) or emitted (+). The functions ω and δ have the same domain, while the domains of γ and δ are disjoint. Note that this conditions guarantee the resulting automaton to be deterministic, as stated by the following rules.

if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$ and $\omega(a, q) = +$

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \qquad \text{(E-StepA)}$$

if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$ and $\omega(a, q) = -$

$$(\sigma, q) \xrightarrow{\tau}_E (\sigma', q') \qquad \text{(E-StepS)}$$

if $\sigma = a; \sigma'$ and $\gamma(a, q) = (b, q')$

$$(\sigma, q) \xrightarrow{b}_E (\sigma, q') \qquad \text{(E-Ins)}$$

otherwise

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \qquad \text{(E-Stop)}$$

Also note that the single-step rules can be generalised to sequences of actions by computing the transitive closure of the above transitions.

We can observe that the class of security properties defined through gate automata can be mapped into edit policies.

Proposition 4.1 For each gate automaton $\mathcal{G}$ there exists an edit automaton $E_{\mathcal{G}}$ enforcing the same property of $\mathcal{G}$.

Proof. We start from a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ and we instantiate it to the interface automaton $\mathbf{G}$. We denote the input (output) symbols of $\mathbf{G}$ with $\alpha?$ ($\alpha!$). Then build the two sets[3]

- $Q^V = \{q^v \mid v \in V\}$ and

[3]Note that, even though the reaction sequences σ can be infinite, i.e. $v \overset{\sigma!}{\underset{\alpha?}{\Longrightarrow}}\uparrow$, we just need a finite number of states for representing them. Indeed, applying the pumping lemma for regular languages to (the definition of instantiation of) our gate automata we observe that each infinite reaction sequence σ can be represented as $\sigma = \sigma_1\sigma_2^*$ where $*$ is the Kleene star and both σ_1 and σ_2 are finite. However, as the infiniteness of Q^Σ does not affect the proof, we can still imagine it as an infinite set.

- $Q^{\Sigma} = \bigcup\{q^{\sigma}_{v,\alpha} \mid \exists \sigma'.v \overset{\sigma'!}{\underset{\alpha?}{\Longrightarrow}} \wedge \cdot \preceq \sigma \preceq \sigma'\}$

Now we define the functions ω and δ as follows.

- for each v, α such that $v \xrightarrow{\alpha}$ and $v \overset{\sigma!}{\underset{\alpha?}{\Longrightarrow}}$
  - $\omega(\alpha, q^v) = -$
  - $\delta(\alpha, q^v) = q^{\sigma}_{v,\alpha}$
- for each v, α such that $v \overset{\alpha}{\not\rightarrow}$
  - $\omega(\alpha, q^v) = +$
  - $\delta(\alpha, q^v) = q^v$

Finally we define γ as:

$$\gamma(\_, q^{\beta\sigma}_{v,\alpha}) = (\beta, q^{\sigma}_{v,\alpha})$$

where u is a state such that $v \overset{\sigma'!}{\underset{\alpha?}{\Longrightarrow}} u$ for some σ', $\_$ is any possible action symbol (also including $\cdot$) and $q^{\cdot}_{v,\alpha} = q^u$.

We can easily verify that $E_{\mathcal{G}} = \langle Q^V \cup Q^{\Sigma}, q^{\imath}, \delta, \gamma, \omega \rangle$ is a valid edit automaton. Indeed, by construction $Dom(\omega) = Dom(\delta) = Q^V \neq Dom(\gamma) = Q^{\Sigma}$ and $Q^V \cap Q^{\Sigma} = \emptyset$.

The last part of the proof consists of showing that for each finite input trace $\mathcal{G}$ and $E_{\mathcal{G}}$ produce the same output. We show that for each action α the two automata $\mathbf{G}$, i.e. an instantiation of $\mathcal{G}$, and $E_{\mathcal{G}}$ starting from the states v and q^v generate the traces $\sigma!$ and σ. Moreover, we show that if $\mathbf{G}$ and $E_{\mathcal{G}}$ do not diverge then they reach the states u and q such that $q = q^u$.

For each α two cases arise: either $v \overset{\alpha}{\not\rightarrow}$ or $v \xrightarrow{\alpha} v'$.

- If $v \overset{\alpha}{\not\rightarrow}$ then, by definition of $\mathbf{G}$, $v \overset{\alpha!}{\underset{\alpha?}{\Longrightarrow}} v$. Moreover, by construction of $E_{\mathcal{G}}$, $\omega(\alpha, q^v) = +$ and $\delta(\alpha, q^v) = q^v$ from which the thesis follows.
- If $v \xrightarrow{\alpha} v'$ then we have two sub cases:
  - $v \overset{\sigma!}{\underset{\alpha?}{\Longrightarrow}} u$. In this case, $\omega(\alpha, q^v) = -$ and $\delta(\alpha, q^v) = q^{\sigma}_{v,\alpha}$. Then δ applied to $q^{\sigma}_{v,\alpha}$ generates each action of σ till reaching q^u.
  - $v \overset{\sigma!}{\underset{\alpha?}{\Longrightarrow}}\uparrow$. We proceed similarly to the previous case. Just note that, as σ is infinite, δ cannot lead to a state q^u for each $q^u \in Q^V$.

□

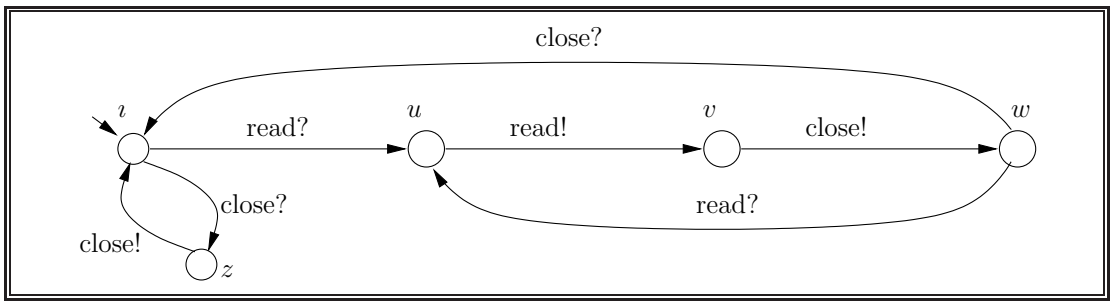


Figure 4.27: The instantiation of the *file closing* policy.

Example 4.20 Consider again the automaton of Example 4.18. We instantiate it to the interface automaton of Figure 4.27

Applying the procedure described in Proposition 4.1 we generate the edit automaton defined by the following states and functions (for brevity we only report the relevant cases).

$$
\begin{array}{lll}
Q^V = \{q^\iota, q^u, q^v, q^w\} & & \\
Q^\Sigma = \{q^{\cdot}_{\iota,\mathrm{read}}, q^{\mathrm{read}}_{\iota,\mathrm{read}}, q^{\mathrm{read;close}}_{\iota,\mathrm{read}}\} \cup \{q^{\cdot}_{\iota,\mathrm{close}}, q^{\mathrm{close}}_{\iota,\mathrm{close}}\} & & (q_\iota) \\
\quad \cup \emptyset \cup \emptyset & & (q_u) \\
\quad \cup \emptyset \cup \emptyset & & (q_v) \\
\quad \cup \{q^{\cdot}_{w,\mathrm{read}}, q^{\mathrm{read}}_{w,\mathrm{read}}, q^{\mathrm{read;close}}_{w,\mathrm{read}}\} \cup \{q^{\cdot}_{w,\mathrm{close}}\} & & (q_w) \\
\quad \cup \emptyset \cup \emptyset & & (q_z)
\end{array}
$$

ω	δ	γ
$\omega(\mathrm{close}, q^\iota) = -$	$\delta(\mathrm{close}, q^\iota) = q^{\mathrm{close}}_{\iota,\mathrm{close}}$	$\gamma(\_, q^{\mathrm{close}}_{\iota,\mathrm{close}}) = (\mathrm{close}, q_\iota)$
$\omega(\mathrm{read}, q^\iota) = -$	$\delta(\mathrm{read}, q^\iota) = q^{\mathrm{read;close}}_{\iota,\mathrm{read}}$	$\gamma(\_, q^{\mathrm{read;close}}_{\iota,\mathrm{read}}) = (\mathrm{read}, q^{\mathrm{close}}_{\iota,\mathrm{read}})$
$\omega(\mathrm{close}, q^w) = -$	$\delta(\mathrm{close}, q^w) = q^{\cdot}_{w,\mathrm{close}} = q^w$	$\gamma(\_, q^{\mathrm{close}}_{\iota,\mathrm{read}}) = (\mathrm{close}, q_w)$
$\omega(\mathrm{read}, q^w) = -$	$\delta(\mathrm{read}, q^w) = q^{\mathrm{read;close}}_{w,\mathrm{read}}$	$\gamma(\_, q^{\mathrm{read;close}}_{w,\mathrm{read}}) = (\mathrm{read}, q^{\mathrm{close}}_{w,\mathrm{read}})$
		$\gamma(\_, q^{\mathrm{close}}_{w,\mathrm{read}}) = (\mathrm{close}, q_w)$

Let now consider again the trace of Example 4.18, i.e., $\sigma = \mathrm{read};\mathrm{close}$. Applying the edit automaton defined above, we obtain

$$(\mathrm{read};\mathrm{close}, q^\iota) \xrightarrow{\tau}_E (\mathrm{close}, q^{\mathrm{read;close}}_{\iota,\mathrm{read}}) \xrightarrow{\mathrm{read}}_E (\mathrm{close}, q^{\mathrm{close}}_{\iota,\mathrm{read}}) \xrightarrow{\mathrm{close}}_E (\mathrm{close}, q_w) \xrightarrow{\tau}_E (\cdot, q^\iota)$$

which corresponds to the application of the rules (E-StepS), (E-Ins), (E-Ins) and (E-StepS).

As the definition of edit automaton poses no restriction on how the functions ω and γ are computed, the opposite of Proposition 4.1 is not true in general.

4.3 S×C×T through gate automata

In this section we present the enforcement environment based on our gate automata. As we saw in Section 4.1, the S×C×T runtime enforcement relies on the retrieval of trust feedback from the running programs. Also, it must be able to change its configuration according to the trust values. Here we discuss how gate automata satisfy both these requirements.

We describe how gate automata drive the enforcement mechanism in a security framework based on S×C×T. This approach represents an extension of the original S×C×T model [64] in which the policy enforcement is limited to target truncation. Moreover, we improve S×C×T, where the trust management is triggered only by contract violations, by integrating trust-oriented actions in the policies specification.

4.3.1 Gate Automata and ConSpec

We saw in Section 3.3 how *ConSpec* has been proposed as a formalism for defining both behavioural contracts and security policies. Here we recall the syntax of ConSpec and we show how ConSpec specifications can be translated into corresponding gate automata. Note that, for simplicity, we omit a few details of the original ConSpec syntax irrelevant for our purposes.

Roughly, a ConSpec specification is composed by three blocks: (i) a preamble, (ii) a security state and (iii) a finite list of clauses. The preamble just declares the range of values for the used variables (`MAXINT` and `MAXLEN`)[4]. The security state is a list of variables declarations following the schema τ `x` ::= `v` where $\tau \in \{\texttt{bool}, \texttt{int}, \texttt{string}\}$ is a type, `x` is a variable name and `v` is a value of type τ. Note that, following the definition given in [7], types are bounded, i.e., they represent a finite number of values. For instance, if we set `MAXINT` to 3 then integer values range in $\{0, 1, 2, 3\}$.

Each clause contains a parametric action $\alpha(\tau\ y)$, activating the rule, and a list of conditional instructions. Action names belong to a denumerable set Λ, i.e., $\alpha \in \Lambda$, and types are the same as for the security state. The left side of the conditional instructions is a decidable, boolean guard g defining a property of the security state and action parameter, while the right side is an update statement u (i.e., a possibly empty block of variable assignments). We assume all the guards of a single clause to be pairwise disjoint, i.e., if g and g' belong to the same clause then it never happens that $g \wedge g'$ is verified. Figure 4.28 shows the syntax described above.

The structure of security clauses requires some further explanations. Indeed, comparing it with the standard one [6, 7], we see two main differences: (i) we only have before-event checks (i.e., we do not use the keywords `AFTER` and `EXCEPTIONAL`) and (ii) we use monadic actions. We claim that these simplifications do not reduce

[4]The standard ConSpec syntax also contains statements defining the scope of a policy, i.e., `Session`, `Multisession` and `Global`. However, it is immaterial for our purposes and we can simply neglect it.

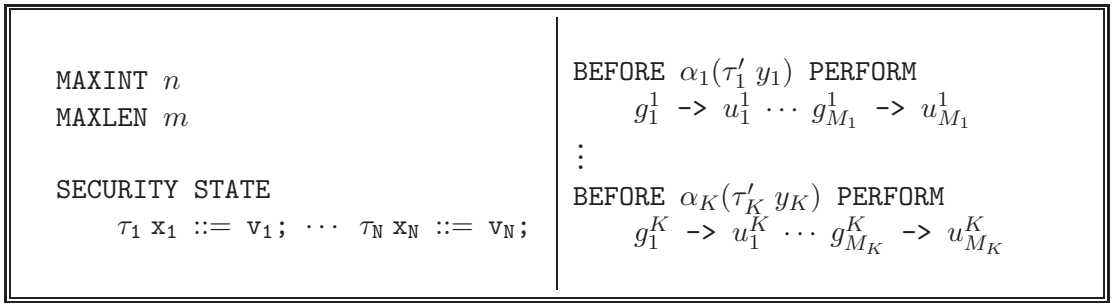


Figure 4.28: The ConSpec preamble, security state (left) and clauses (right).

the expressive power of the ConSpec language. As a finite number of parameters can be encoded in a single one, using monadic actions is not a restriction. For instance, we could use strings to encode n-arguments actions (e.g., α("3,msg,false") for $\alpha(3,$"msg", $false)$). Then, we can use the string operations in the guards of the rules for extracting the actual parameters. Also, we require all the variable and parameter names to be unique and all the clauses to be triggered by different actions.

Moreover, we can simulate the behaviour of AFTER and EXCEPTIONAL clauses by introducing new actions. As a matter of fact, the standard syntax of ConSpec aims at modelling the computations of object-oriented systems, i.e., passing through method invocations. Every method triggers the clauses when it is invoked, when it returns a result and, possibly, when raising an exception. Then, for each method m we can define three actions $\alpha^B_{\mathtt{m}}$, $\alpha^A_{\mathtt{m}}$ and $\alpha^E_{\mathtt{m}}$ representing the method invocation, standard return and exceptional return, respectively.

Example 4.21 Consider the policy saying "An application cannot open connections after reading local files". We model the involved methods through the actions fopen(int mode) and copen(string url). Where mode $\in \{0,1,2,3\}$ is a two-bits mask representing the access type (i.e., 00 = none, 01 = read, 10 = write and 11 = read and write), and url is a network address. The resulting policy is:

```
MAXINT 3
MAXLEN 0
SECURITY STATE
bool accessed ::= false;

BEFORE fopen(int mode) PERFORM
  (mode == 1) -> {accessed ::= true;}
  (mode == 3) -> {accessed ::= true;}
  (mode == 0 || mode == 2) -> {}
BEFORE copen(string address) PERFORM
  !accessed -> {}
```

The semantics of ConSpec can be given using gate automata. Given a state q

and a guard g, we say that g is valid in q ($q \vdash g$) if and only if replacing the variable names of g using the mapping defined by q we obtain a tautology. Moreover, we say that an update block u denotes a function, namely $[\![u]\!]$, from states to states, i.e., $[\![u]\!] : Q \to Q$.

We obtain a gate automaton from a ConSpec specification as follows.

States. The set Q of states is fully characterised by the security state and the actions parameters. In particular, we define a state q as a mapping from variable and parameter names to the lifted domain of possible values. Formally, given a variable or parameter name x, then $q(x) = v$ with $v \in Val \cup \{\bot\}$ (where $Val = \texttt{int} \cup \texttt{bool} \cup \texttt{string}$). Moreover, to be valid a state must assign to each variable a value different from $\bot$ and to at most one parameter a value that is different from $\bot$. Hence, Q is the set of all the possible, valid combinations of assignments. Note that, as ConSpec uses bounded types, the number of states is always finite.

Initial state. The initial state $\imath \in Q$ is the set mapping the variables of the security state to their initial values and the parameters to the undefined, $\bot$ value.

Alphabet. The set of events A that the automaton can read is the set of pairs $\{\langle \alpha, v\rangle \mid \alpha \in \Lambda \wedge v \in Val\}$. We use $\alpha(v)$ instead of $\langle \alpha, v\rangle$ where unambiguous.

Transitions. We build the set T of transitions in the following way. For each ConSpec clause we take the triggering action $\alpha(\tau\, x)$ and we list all the states $q \in Q$ such that $q(x) = \bot$. Then we proceed as follows.

1. For each possible event $\alpha(v)$ we add a transition from $q \xrightarrow{\bar{\alpha}(v)} q'$, where $\forall y \neq x. q'(y) = q(x)$ and $q'(x) = v$.
2. For each conditional instruction $g \to u$ of the clause and for each of the freshly added transitions, if $q' \vdash g$ then we add a transition $q' \xrightarrow{\alpha(v)} [\![u]\!](q)$.
3. For all the states $\dot{q}$ such that $x = \bot$ and for all the events $\alpha(\dot{v})$ such that $\dot{q} \not\xrightarrow{\bar{\alpha}(\dot{v})}$, we add a transition $\dot{q} \xrightarrow{\bar{\alpha}(\dot{v})} \dot{q}$.

We iterate these steps until every clause has been processed.

Example 4.22 We create a gate automaton for the specification in Example 4.21.

Figure 4.29 shows the gate automaton produced by the procedure described above. Rows and columns denote the values of variables for the automaton states, for instance the top row contains the states q such that $q(\texttt{access}) = \texttt{false}$. The left-most column contains the states assigning no values to the actions parameters. The unreachable state in position $\texttt{access} = \texttt{true}, \texttt{url} = ""$, which would correspond to a specification violation, has been removed. Also, two pairs of column, i.e., $\texttt{mode} = \texttt{0/2}$ and $\texttt{mode} = \texttt{1/3}$, have been grouped as their states share the same behaviour. Finally, we did not draw immaterial self loops, i.e., representing transitions that cannot take place.

Clearly, the procedure described above can be optimised in several ways, e.g., removing unreachable states or collapsing groups of equivalent states. Nevertheless,

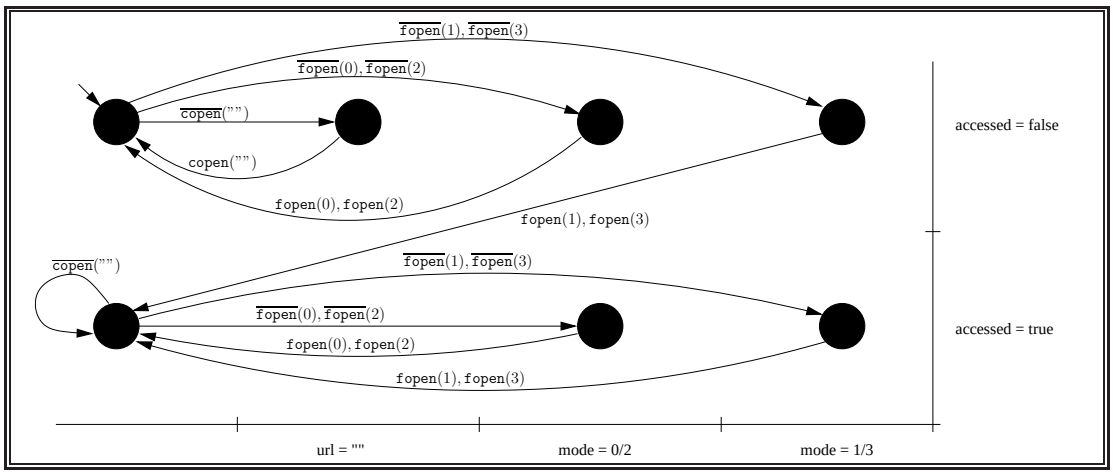


Figure 4.29: The conversion of a ConSpec specification into a gate automaton.

our purpose is to show that gate automata can be suitably used to encode ConSpec policies and contracts.

Note that the opposite direction is not possible according to the syntax of ConSpec given in Figure 4.28. In other words, we cannot translate a gate automaton in a ConSpec policy without introducing some modification. First of all, we should introduce facilities for updating the trust values. This could be done by extending the syntax and semantics of the ConSpec rules.

Another modification would be needed for allowing the execution of security-relevant methods, i.e. reactions. Again an extension of the syntax would be needed. However, this would raise some issues about the semantics of the composition of two specifications. Indeed, we should discuss whether the actions generated by one policy should be also processed by the others and in which order. In the next section we will show how gate automata can be hierarchically organised for dealing with these issues.

4.3.2 Enforcement environment

The S×C×T workflow, depicted in Figure 4.20, shows the two phases of the application deployment process: the evaluation of trustworthiness and the assignment to a security domain. When an application enters the deployment procedure, i.e., before its first execution, the trust module decides about the trustworthiness of the code provider. This amounts to accept the truthfulness of the contract and its source.

If this check is not passed, i.e., the system rejects the vendor's trustworthiness, then the application runs in the scope of the *policy enforcement* mechanism. Otherwise, if the trust check succeeds, the system checks whether the contract complies with the security policy. In case of compliance, the system executes the application under a *contract monitoring* setting. While the policy enforcement process

prevents the security violations, the monitoring facility keeps under control the possible contract violations. When a running program violates its contract, i.e, it tries to perform in an undeclared way, the system reacts by changing the trust level of the application provider.

Here we introduce an implementation of the S×C×T runtime support using gate automata. According to the S×C×T standard model [64, 65], applications run in the scope of one of the two security domains described above. In both cases, running programs are dynamically checked for compliance with respect to their contract (i.e., contract monitoring process). Moreover, the applications watched by the policies enforcement facility are checked for possible policy violations.

The platform owners declare their security policies through gate automata either directly or translating ConSpec policies (see Section 4.3.1). Instead, we assume that the contracts are always specified through ConSpec.

Starting from a ConSpec contract, we build a corresponding gate automaton by following the procedure for policies presented in the previous section. The only difference is that here we replace the third step of the transitions creation procedure with

3. For all states $\dot{q}$ s.t. $x = \perp$ and for all events $\alpha(\dot{v})$ s.t. $\dot{q} \stackrel{\bar{\alpha}(\dot{v})}{\not\longrightarrow}$, we add a fresh, new state $q^\star$ in Q and a pair of transitions $\dot{q} \stackrel{\bar{\alpha}(\dot{v})}{\longrightarrow} q^\star$ and $q^\star \stackrel{\blacktriangledown}{\rightarrow} \dot{q}$ in T.

In this way, as expected, a contract violation leads to a trust penalty. This behaviour implements the S×C×T reaction to the contract violations.

We use the gate automata specifications of policies and contracts for implementing the S×C×T runtime environment. We consider a program R as a source of the security-relevant actions, that are the side effects of the programs' executions. Moreover, we assume the enforcement environment to be effective, i.e., R can be suspended before the actual execution of the operation corresponding to the ongoing action. For instance, if R tries to access a resource, so raising an access action, it actually obtains the permission only after checking the security settings.

The first component of the enforcement environment is the *trust management system* (TMS). This component handles the trust weights associated to each agent and provides an implementation of the two internal actions ▲ and ▼. While following the execution of its target, the enforcement environment can perform one or more actions of type ▲ and ▼. The TMS receives these signals and increases (decreases) the target trust level. Note that some TMSs use a finer characterisation of rewards and penalties, i.e., more than two actions. Nevertheless, this behaviour is fully compatible with our model. Indeed, we can easily extend the set of internal actions or simulate it by adding more consecutive transitions.

The enforcement environment also contains a set of gate automata $\mathcal{G}^1, \ldots, \mathcal{G}^n$ composing the *policy pool* (PP). The automata in the policy pool are associated to a certain level of trust $0 \leqslant t \leqslant 1$ on which they are inversely ordered, i.e.,

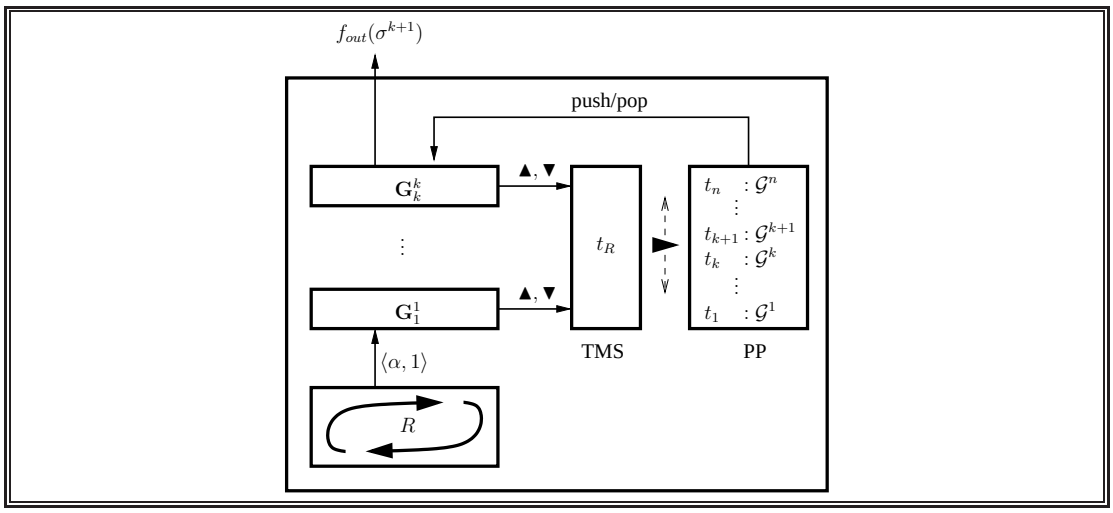


Figure 4.30: The enforcement environment based on gate automata.

$1 \leqslant i < j \leqslant n$ implies that $t_i > t_j$. We also insert the gate automaton obtained from the contract of R in PP. The level of trust of this automaton is always equal to 1 and it is the first in the ordering.

When a target R, having trust level t_R, starts its execution, the policy pool instantiates all the gate automata $\mathcal{G}^i$ such that $t_i \geqslant t_R$ to the corresponding interface automata $\mathbf{G}_i^i$ (see Section 4.2.2). Then, the resulting interface automata are composed to create an *interface automata stack* that is applied to R. Note that the automaton obtained from the contract of R is always in the first position of the stack, i.e., the stack bottom.

The stack receives the actions performed by R and processes them by passing the reaction sequences of each automaton to the layer above. More in detail, assuming that the current state of each interface automaton $\mathbf{G}_i^i$ is v_i, every layer of the stack follows this procedure:

1. $\mathbf{G}_i^i$ receives a trace σ^i from the level below;
2. for each element $\langle \bullet, i \rangle$ of σ^i execute the following sub steps:
   (a) if $\bullet = ▲$ ($▼$) then require the TMS to increase (decrease) t_R.
   (b) otherwise, if $\bullet = \alpha$ compute $v_i \overset{\sigma^{i+1}}{\underset{\langle \alpha, i \rangle}{\Longrightarrow}} v_i'$ and pass the control to the layer above (by invoking this procedure);
3. return the control to the level below.

When R fires some action α, the previous steps are executed starting from the first layer, representing the contract of R, with $\sigma^1 = \langle \alpha, 1 \rangle$. The output of the last layer

(after removing the index k) is a sequence of reactions that have been stimulated by α, that is, the enforcement result.

As the actions pass through the stack levels, the TMS receives trust adjustment signals. As a consequence, the TMS updates t_R, possibly causing the system to add or remove one or more automata in the stack.

Figure 4.30 depicts the environment described above. We used $\mathcal{G}^1$ to denote the gate automaton obtained from the contract of R.

4.4 Discussion

In this chapter we described our proposal for an integrated security and trust paradigm. In particular, the chapter went through the following topics.

- Section 4.1 details the Security-by-Contract-with-Trust model and its features.
- Section 4.2 introduces gate automata as an integrated formalism for specifying security and trust policies.
- Section 4.3 proposes a strategy for implementing the S×C×T framework through the application of gate automata.

At the current stage, the relations between security and trust properties are still under investigation. Some simulations have been carried out in [64, 65]. This preliminary analysis shows that our approach is technically feasible and a prototype is currently under implementation. However, conclusive results have not been produced yet.

The complexity of studying and providing security assurances on the behaviour of large scale, distributed systems can be a serious issue during the design and development of software and services. Many modern systems opted for trust-based solutions. Concepts like "trust" and *reputation* proved to be more than sufficient for mitigating security risks in many practical cases. We think that security analysis and trustworthiness evaluation systems can be integrated to obtain hybrid solutions exploiting both the approaches.

Secure Service Composition

This chapter presents our work on *secure service composition*. Service composition is crucial in Service-Oriented Computing (SOC). As a matter of fact, almost every functionality of a service network (directly or indirectly) depend on how the services compose each other. Composition requests are declared in the implementations of a service at development time. This means that the development framework must offer to the programmes the facilities for performing services invocations. After the deployment of the service, the invocations must be interpreted, according to the actual network structure, for producing the real composition.

The dynamic composition includes two stages, i.e., *choreography* and *orchestration* [154]. Choreography identifies the end-to-end composition between two services. Many of the relevant aspects of service choreography consider the cooperation rules, e.g., the sequence of the exchanged messages and their content. Instead, orchestration deals with the composition of multiple services in terms of the business process they generate. Here we mainly focus on the security aspects of the service orchestration. From the security perspective, the orchestration process should be carried out respecting the security requirements (i.e., the policies) of the parts involved in the composition. As long as the service orchestration is performed automatically, security verification steps must be included in the procedure.

Our approach consists in proposing a way of supporting the automatic verification of the service orchestration. We start in Section 5.1 by introducing the model that we used for defining services and security policies. Than, we proceed with the presentation of our framework for the safe composition of orchestration plans (Section 5.2). Finally, in Section 5.3 we conclude by extending our system with a mechanism for declaring security prerequisites that services apply to the client sessions in which they are involved.

5.1 Security issues in open networks

Major work has been done for investigating the interactions between a service and its clients and how to secure them. As a consequence of these efforts some security standards have been defined. For instance, WS-SecurityPolicy [40, 75] provides an XML-based syntax for specifying and enforcing security requirements over services executions. The effort devoted to the creation of standards for the security aspects

of the service networks outlines the interest of researchers and industries in finding reliable security mechanisms for the web services.

Recently, Bartoletti et al. [23], outlined the importance of extending the security requirements to the whole network structure. In this section we extend the security model of [23] to the *open networks*, i.e. service networks having some unspecified components.

5.1.1 Open networks

Many theoretical frameworks dealing with the security issues of service networks rely on a theoretical model representing the network or a part of it. For instance, several authors proposed models based on the notion of *session*, e.g., see [39, 45, 50, 117]. Generally speaking, a session represents the temporary composition of two or more services. These models provide a pure framework allowing for a simpler and more elegant investigation of the properties of the networks, e.g. through static verification. However, having a static knowledge of all the participants composing a service network or a complete session can be a quite strong assumption.

Service networks have been proposed for dealing with highly dynamic environments based on pervasive interoperability [147]. Even though it seems reasonable that a service developer knows at least a part of the existing services, a prediction of all the possible sessions that might involve his own service is often unrealistic. Nevertheless, we would prefer to maintain the advantages deriving from the static verification of sessions.

Here, we move to different assumptions. We use *open sessions* to model the, intrinsically incomplete composition of a group of services in an *open network*. Open networks can model the behaviour of service networks without assuming a static knowledge of the whole network structure. Below we present the advantages of using open sessions and we use them to model a real-world case study.

Motivations. Service networks are meant to be distributed, large scale systems. Their complexity poses a barrier to using static techniques for verifying their security properties. Indeed, even assuming non-circularity in service invocations, that is the requests chains are always terminating, the number of possible compositions grows rapidly with the total number of services in the network.

A further issue derives from the incremental development of service networks. In Section 3.1 we saw how a security verification step can be included in the software development process. However, a similar solution is not viable for web services. As a matter of fact, each service is developed independently from the others and it can join an existing network in any moment. Thus we cannot expect to reduce all the static analysis operations to a single step of the development process.

Nevertheless, the security properties of service networks can be suitably verified using static approaches. Also consider that, due to the distributed deployment of

the service networks, dynamic monitoring techniques are difficult to apply properly. Monitoring systems for web services have been proposed in the literature, e.g. see [171, 165, 19, 92] However, as each service can run on its own physical platform many visibility issues arise. Moreover, the idea of a centralised monitor watching (part of) the network seems to be in contrast with purpose of a distributed environment. Hence, many authors (e.g. see [82, 43, 51]) advocate contract-based approaches to be the reference models for the security verification of service compositions and orchestrations.

Bartoletti et al. [23] proposed a language-based approach for defining security policies directly inside the instructions of a service implementation. Their model only deals with closed networks, i.e. with services whose components are fully specified a priori. The resulting composed service is indeed dealt with as a single, large scale service. However, services are incrementally built, and components may appear and disappear dynamically. It is then important to also analyse *open* networks, i.e. networks having unspecified participants. As a matter of fact, service-oriented paradigms aim at guaranteeing compositional properties and their behaviour should be independent from the actual implementation of (possibly unknown) parties. Moreover, closed networks orchestrated by a global composition plan require to be completely reorganized whenever a service fails or a new one becomes available.

The main contribution of this section is extending the results of [23] to open networks. In particular, we will define *partial* plans that only involve parts of the known network, i.e. open sessions, and that however will be safely adopted within any operating context. We also show how these partial plans can be combined together, along with the composition of the services they come from. As expected, composability of viable plans is subject to some constraints, and we also outline a possible way to efficiently check when these constraints are satisfied.

Furthermore, we enrich the contract that a service offers with the requirements a client must fulfil to be served. This extension accounts for when the execution of the service is affected by that of the client, e.g. being already registered in a digital identity list. We propose a procedure for synthesizing these requirements for all the services that compose a given open network. Note that the requirements generally depend on all the network components. Hence, to guarantee security we need a composition plan for computing these requirements and therefore a service cannot always specify them safely in isolation.

Policy scope. In [23] the security policies are applied to a local scope, i.e. a limited block of instructions being part of a service implementation. The scope is defined at development-time through a proper language operator (see Section 5.1.2). Informally, the meaning of the policy scope is that the policy must be respected while executing the internal instructions. Dynamically, this amount to say that, within its scope, the policy is checked against the current execution trace before each computational step (involving security aspects).

From a traditional perspective, this method correctly apply for modelling the security policies of a program running on a single platform (see Section 3.1). However, some further discussion is needed for web services. Indeed, local policies correspond to temporal properties defined on the sequence of security-relevant operations, i.e., the execution history, performed at runtime. While a single application starts its execution from an empty trace, a web service is usually invoked after a previous computation which was carried out elsewhere.

Closed networks simplifies this issue by assuming that all the computational units executing before the invocation of a service are statically defined. In this way, when entering the scope of a policy, a set of possible execution history is given. This set is safe in the sense that it always contains the history that will actually take place. In this way, the local policies have a well defined security context on which they can be evaluated.

As we aim at releasing this assumption, i.e. the existence of a static approximation of the whole network, we need to redefine the behaviour and scope of the local policies. In particular, here we assume that the validity of a local policy only depends on the computation taking place within its scope. This means that, when entering a policy scope, the policy evaluation starts from an empty history. We say that these policies are *hyper-local* in order to distinguish them from the standard local policies that we used so far, e.g. in Section 3.1. As in this section we always refer to them, for brevity we simply use the expression "local policies".

Still, hyper-local policies can be suitably used for defining many security properties of interest. Moreover, they apply pretty well for modelling the call-by-contract security interactions among web services. Indeed, service invocations can lay in the scope of a security policy. In this way, the security requirements only flow from the the caller to the callee. We discuss this aspect and its consequences in Section 5.1.2 when introducing our operational semantics for web services.

We also introduce the notion of *security prerequisite* in our model. Being applied to the execution history preceding a service invocation, security prerequisites complement hyper-local policies. A prerequisite is defined through the same formalism, i.e. usage automata, as a local policy. We use a partial evaluation technique for automatically synthesising prerequisites starting from a usage automaton and a service implementation. This part will be discussed in Section 5.3.

Working example. Below we introduce our working example. We start by giving an informal description of a simple network implementing a travel-booking service. Figure 5.31 shows how the service network is organised. Rounded boxes denote locations that host services. Dashed lines contain locations with homogeneous services, i.e. services offering similar functionalities. Clients contact the travel agency providing a credit card number for payments and receive back a receipt. Every instance of the travel agency books exactly one room and one flight. The responsibility of doing an actual reservation is delegated to booking services. Each booking service

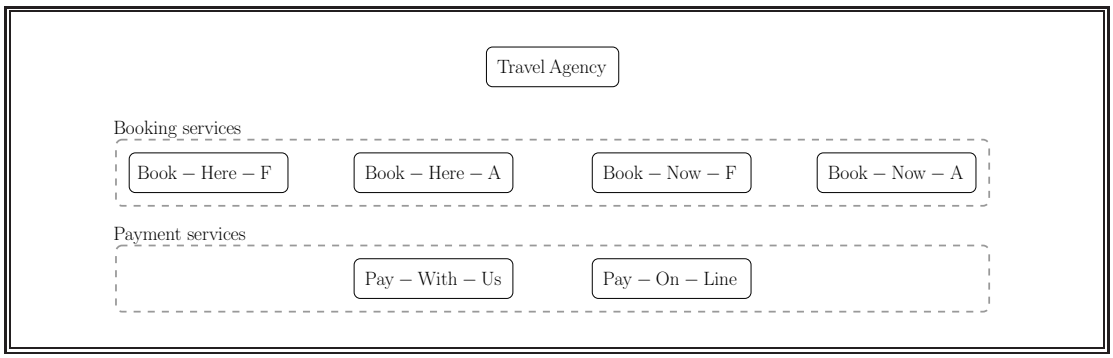


Figure 5.31: A travel booking network

receives a card number and uses it for paying a reservation. Payment services are in charge of authorising a purchase. A payment service charges the needed amount on the credit card (possibly after checking some property of the card number), and returns `TRUE`. Otherwise, it answers `FALSE`.

Moreover, we imagine that each service is interested in declaring and verifying its own security requirements. Clearly, different services focus on different security aspects, depending on the service resources, interactions, context, etc. For example, since a booking service invokes a payment service, it would like to "perform at least one availability check before each payment"; we call this policy $\varphi_{\texttt{BN}}$. Another possible policy for an efficient booking service says "never execute unnecessary checks", referred to as $\varphi_{\texttt{BH}}$. Similarly, the travel agency can declare rules concerning the intended behaviour of the booking services it wants to use. A typical policy of a travel agency might be "never book twice the same service (accommodation or flight)", call it $\varphi_{\texttt{TA}}$. Note that this last policy also involves a functional requirement. Indeed, the correct behaviour of the travel agency consists in booking exactly one flight and one hotel. Our formalism is expressive enough to represent both functional and non functional constraints in our policies.

In this example, the clients of the travel agency are left unspecified, so the network is *open*, as one or more components are missing. As a matter of fact, clients cannot affect at all the security policies introduced so far. We can therefore check whether this simple network satisfies the constraints specified by the involved services, regardless of the presence of any clients.

Even though in most cases services do not put explicit security constraints on their clients, as done here, sometimes it would be helpful to expose policies governing service usages. This happens in reality. For example, new clients are often required to register before being allowed to access a service. Example 5.34 considers this case, further detailed in Section 5.3.2, where we address this more general situation.

5.1.2 Service structure

Our programming model for service composition is based on λ^{req}, which was introduced in [23]. The syntax of λ^{req} extends the classical call-by-value λ-calculus with two main differences: *security framing* and call-by-contract *service request*. Syntactically, a security framing embraces a term and it represents the scope of a security policy. Instead, a request denotes the invocation to a remote service. In this section, we provide the reader with a detailed description of the λ^{req} syntax.

Service networks are sets of services. A service e is hosted in a location ℓ, e.g. denoting a network address. We assume that there exists a trusted, public *service repository* $\mathtt{Srv}$ that contains references to available services. Abstractly, an element of $\mathtt{Srv}$ has the form $e_\ell : \tau \xrightarrow{H} \tau'$, where e_ℓ is the code of the service, ℓ is the unique location that hosts the service, $\tau \to \tau'$ is the type of e_ℓ and H is its effect. Types represent a functional signature of the service in terms of input/output values. Clients requiring a service must specify its type and $\mathtt{Srv}$ returns an instance satisfying it. Instead, an effect H represents the behaviour of the associated service mainly expressing the security-relevant events. In other words, the effect H provides the clients with a behavioural contract of the side effects produced by a service that may affect security.

Usage policies. A usage policy governs the access to resources that we may wish to protect. A policy φ is defined through a corresponding *usage automaton* A_φ. Usage automata are much like non-deterministic finite state automata (NFA). Briefly, a usage automaton consists of an input alphabet of events Ev, a finite set of states Q, an initial state $\imath$, a set of final states F and a set T of transitions labelled by events.

In order to define the events, we assume as given a denumerable set of variables Var, ranged over by $x, y, ...$, a finite set of resources Res, ranged over by $\{r, r', r_1, ...\}$, a finite set of actions Act, ranged over by $\alpha, \beta, \ldots$ These sets are pairwise disjoint. Also, we let $\dot{x}, \dot{y}$ range over $\mathsf{Res} \cup \mathsf{Var}$. Then, the events belong to $\mathsf{Ev} \subseteq \mathsf{Act} \times (\mathsf{Res} \cup \mathsf{Var})$, ranged over by $\alpha(\dot{x}), \beta(\dot{y}), \ldots$

The formal definition of usage automaton in given in Table 5.7.

Transitions are labelled with an event $\alpha(\dot{x})$, and we feel free to write $q \xrightarrow{\alpha(\dot{x})} q'$ instead of $\langle q, \alpha(\dot{x}), q' \rangle$). We say that usage automata are parametric over resources, because $\dot{x}$ can be a variable that will eventually be bound to an actual resource, so giving rise to an actual policy. Differently from [28], here we will allow for partial instantiations, because we wish to deal with open systems, and so some resource can be still unknown. A usage automaton is (partially) instantiated through a *name binding function* or *substitution*

$$\sigma : \mathsf{Res} \cup \mathsf{Var} \to \mathsf{Res} \cup \mathsf{Var}, \quad \text{such that } \forall\, r \in \mathsf{Res}.\, \sigma(r) = r$$

We understand that σ is homomorphically applied to the usage automaton A_φ. We

A *usage automaton* A_φ is a 5-tuple $\langle \mathsf{Ev}, Q, \imath, F, T \rangle$, where

- Ev is the input alphabet,
- Q is a finite set of state,
- $\imath \in Q$ is the initial state,
- $F \subseteq Q$ is a set of final states,
- $T \in Q \times \mathsf{Ev} \times Q$ is a set of labelled transitions.

Table 5.7: Definition of Usage Automaton.

will sometimes write binding functions according to the following grammar

$$\sigma ::= \{\} \mid \{x \mapsto \dot{y}\} \cup \sigma'$$

where x is a variable and $\dot{y}$ can be either a resource or a variable.

Instantiating an automaton A_φ with σ gives back the automaton $A_\varphi(\sigma)$. Essentially, $A_\varphi(\sigma)$ has the same structure of A_φ but its transitions set $T(\sigma)$ is defined to be

$$T(\sigma) = \{q \xrightarrow{\alpha(\sigma\,\dot{x})} q' \mid q \xrightarrow{\alpha(\dot{x})} q' \in T\}$$

A sequence of access events η violates an instance of a usage automaton if it leads to a final state, also called *offending*. Hence, the accepted language is made of violating traces. So a trace η *violates* φ (in symbols $\eta \not\models \varphi$) whenever there exists a σ such that $A_\varphi(\sigma)$ accepts η; otherwise, we say that η *complies* with φ (in symbols $\eta \models \varphi$).

From a language-theoretic point of view, we say that every instance of usage automata defines the upper bound of a class of regular languages over the parametric events alphabet. In symbols

$$\mathcal{L}(A_\varphi) = \bigcup_\sigma \mathcal{L}(A_\varphi(\sigma)) = \{\eta \mid \exists\, \sigma : \eta \in \mathcal{L}(A_\varphi(\sigma))\}$$

Hence, the compliance check between a trace η and a usage automaton A_φ corresponds to verifying whether $\forall\, \sigma.\, \eta \notin \mathcal{L}(A_\varphi(\sigma))$.

The following property states that instantiating a usage automaton we obtain a new automaton defining a less restrictive policy.

Proposition 5.2 For each usage automaton A_φ, mapping σ and trace η

$$\eta \in \mathcal{L}(A_\varphi(\sigma)) \Longrightarrow \eta \in \mathcal{L}(A_\varphi)$$

Proof. Straightforward from the definition of $\mathcal{L}(A_\varphi)$. □

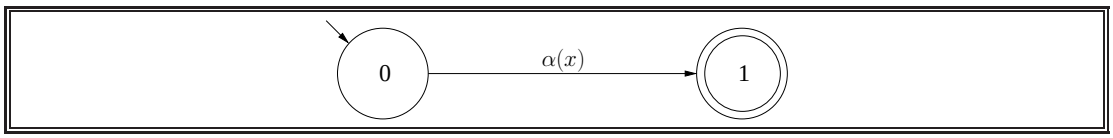
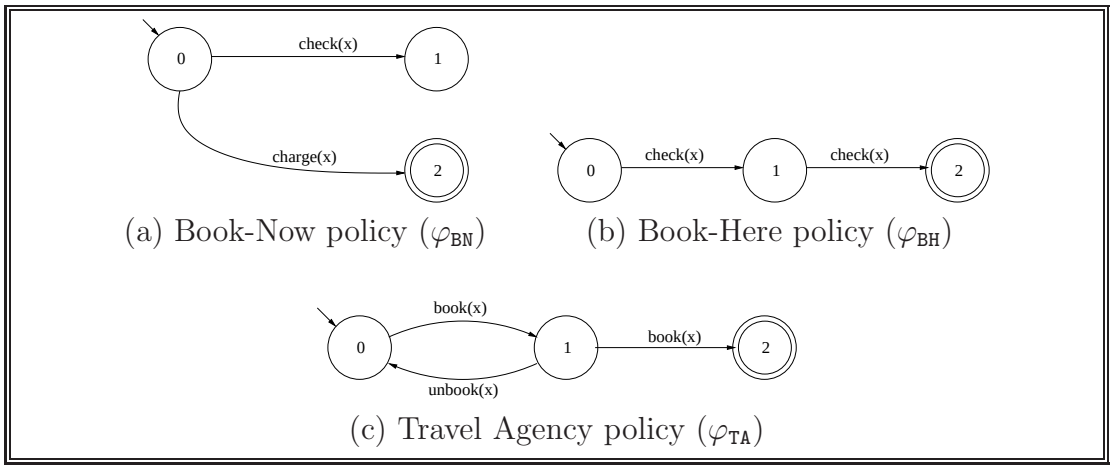


Figure 5.32: A trivial usage automaton (self-loops are omitted).



(a) Book-Now policy ($\varphi_{\texttt{BN}}$) (b) Book-Here policy ($\varphi_{\texttt{BH}}$)

(c) Travel Agency policy ($\varphi_{\texttt{TA}}$)

Figure 5.33: Security policies as usage automata.

Example 5.23 Let us assume $\mathsf{Res} = \{r\}$ and $\mathsf{Act} = \{\alpha, \beta\}$. Now imagine a usage policy φ saying "never $\alpha(x)$", while performing $\beta(r)$ (for some resource r) does not affect the security status. We model this property through a usage automaton with two states, defined as follows

$$A_\varphi = \langle \{\alpha(x), \beta(r)\}, \{0, 1\}, 0, \{1\}, \{\langle 0, \alpha(x), 1\rangle\, \langle 0, \beta(r), 0\rangle, \langle 1, \beta(r), 1\rangle\}\rangle$$

Pictorially, we render this automaton in Figure 5.32. From here onwards, we omit all the immaterial looping transitions, like the ones here labelled $\beta(r)$. Now consider the single-event trace $\alpha(r)$. To prove that this trace is compliant with φ we should verify that it is not accepted by any instance of A_φ. However, the trivial mapping $\sigma = \{x \mapsto r\}$ instantiates A_φ to an automaton that accepts $\alpha(r)$. Instead, any instantiation would leave the automaton in state 0, when checking compliance of the trace $\beta(r)$, which indeed complies with the given policy. □

Example 5.24 Consider now the policies $\varphi_{\texttt{BN}}$, $\varphi_{\texttt{BH}}$ and $\varphi_{\texttt{TA}}$ introduced in Section 5.1.1. The policy $\varphi_{\texttt{BN}}$ is violated whenever a payment (action `charge`) is not preceded by an availability check (action `check`). We model this policy using the usage automaton of Figure 5.33a: starting from the initial state 0, the automaton transits to the *safe* (i.e., non-final pit) state 1 when catching an action `check`. (Recall that immaterial self-loops are omitted in the drawings.) Otherwise, if a `charge`

$e, e' ::=$	$*$	unit
	r	resource
	x	variable
	$\alpha(e)$	access event
	$\texttt{if}\, g\, \texttt{then}\, e \;\; \texttt{else} \;\; e'$	conditional
	$\lambda_z x.e$	abstraction
	$e\, e'$	application
	$\varphi[e]$	security framing
	$\texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$	service request

Table 5.8: The syntax of λ^{req}.

happens, the automaton reaches the final state 2.

Instead, Figure 5.33b shows the automaton for $\varphi_{\texttt{BH}}$. Basically, it counts the number of actions `check` performed on a single resource (identified by x).

Finally, the automaton for $\varphi_{\texttt{TA}}$ is shown in Figure 5.33c. It is similar to that for $\varphi_{\texttt{BH}}$, because both constrain the number of actions (`check` and `book` respectively) performed on the same resource. The only difference it that an action `book` can be cancelled through its "inverse" `unbook`. □

Syntax of services. We introduce in Table 5.8 the syntax of λ^{req}. Similarly to the standard λ-calculus, syntactically correct λ^{req} terms are the closed expressions, i.e. with no free variables, respecting the following grammar. We borrow most constructs from [23] and [28], but for simplicity we do not have a construct for creating resources like in [28].

The expression $*$ represents a distinguished, closed, event-free value. Resources, ranged over by r, r', belong to finite set Res. Access actions α, β operate on resources giving rise to events $\alpha(r), \beta(r'), \ldots$ that actually are side effects.

Function abstraction (where z in $\lambda_z x.e$ denotes the abstraction itself inside e) and application are standard. Note that here we use an explicit notation for conditional branching, the guards of which are defined below. This point will be further clarified in Section 5.1.3. Security framing applies the scope of a policy φ to a program e. Service request requires more attention. We stipulate that services cannot be directly accessed by using a public name or address. Instead, clients invoke services through their public interface, i.e. their type and effect (see Section 5.1.3). A policy φ is attached to the request in a call-by-contract fashion: the invoked service must obey the policy φ. Since both τ and τ' can be higher-order types, we can model simple value-passing interaction, as well as mobile code scenarios. Finally, the label ρ is a unique identifier associated with the request, to be used while planning services.

$g, g' ::= true \mid [\dot{x} = \dot{y}] \mid \neg g \mid g \wedge g'$ ($\dot{x}$,$\dot{y}$ range over variables and resources)

Table 5.9: The syntax of guards.

We use v to denote values, i.e. resources, variables, abstractions and requests. Moreover, we introduce the following standard abbreviations: $\lambda x.e = \lambda_z x.e$ with $z \notin fv(e)$, $\lambda.e = \lambda x.e$ with $x \notin fv(e)$ and $e; e'$ for $(\lambda.e')e$.

The abstract syntax of guards is reported in Table 5.9.

Basically, guards can be either the constant $true$, a syntactic equality check between the variables and/or resources ($[\dot{x} = \dot{y}]$), a negation or a conjunction. All the variables of a guard are bound within it. This implies that the free variables of an expression do not include variables that only occur in guards. We use *false* as an abbreviation for $\neg true$, $[\dot{x} \neq \dot{y}]$ for $\neg\,[\dot{x} = \dot{y}]$ and $g \vee g'$ for $\neg(\neg g \wedge \neg g')$. We also define an *evaluation function* $\mathcal{B}$ mapping guards into boolean values, namely $\{tt, ff\}$, as follows

$$\mathcal{B}(true) = tt \qquad \mathcal{B}([\dot{x} = \dot{x}]) = tt \qquad \mathcal{B}([\dot{x} = \dot{y}]) = ff \quad (\text{if } \dot{x} \neq \dot{y})$$

$$\mathcal{B}(\neg g) = \begin{cases} tt & \text{if } \mathcal{B}(g) = ff \\ ff & \text{otherwise} \end{cases} \qquad \mathcal{B}(g \wedge g') = \begin{cases} tt & \text{if } \mathcal{B}(g) = \mathcal{B}(g') = tt \\ ff & \text{otherwise} \end{cases}$$

Note that $\mathcal{B}$ is total, and that also guards containing variables can be evaluated (according to the second and third rules). In our model we assume resources to be uniquely identified by their (global) name, i.e. r and r' denote the same resource if and only if $r = r'$. In the following, we will use $[\dot{x} \in D]$ for $\bigvee_{d \in D} [\dot{x} = d]$.

Example 5.25 Consider again the services in Figure 5.31. In order to implement them as λ^{req} expressions, we assume that the resources include the following: $Card$, i.e. the set of credit card numbers, with $\mathcal{C}, \mathcal{C}' \subseteq Card$; $Item = \{F, A\}$, i.e. the resources representing flight and accommodation items to book, with $S \in Item$; Rec, i.e. the set of certified receipts, with $rcpt \in Rec$; and the set of boolean resources $Bool = \{\texttt{TRUE}, \texttt{FALSE}\}$.

We now specify the services Travel-Agency, Book-Here-S, Book-Now-S (for brevity we use hereafter -S for either -F or -A), Pay-On-Line and Pay-With-Us through the following corresponding expressions $e_{\texttt{TA}}$, $e_{\texttt{BH}-\texttt{S}}$, $e_{\texttt{BN}-\texttt{S}}$, $e_{\texttt{POL}}$ and $e_{\texttt{PWU}}$.

$$
\begin{aligned}
e_{\texttt{PWU}} &= \lambda x.\texttt{if}\ [x \in \mathcal{C}]\ \texttt{then}\ \texttt{charge}(x); \texttt{TRUE}\ \texttt{else}\ \texttt{FALSE} \\
e_{\texttt{POL}} &= \lambda x.\texttt{if}\ [x \in \mathcal{C}']\ \texttt{then}\ \texttt{check}(x); \texttt{charge}(x); \texttt{check}(x); \texttt{TRUE}\ \texttt{else}\ \texttt{FALSE} \\
e_{\texttt{BH-S}} &= \lambda x.(\lambda y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(S)\ \texttt{else}\ *) \\
&\qquad ((\texttt{req}_{\rho}\, Card \xrightarrow{\varphi_{\texttt{BH}}} Bool)x) \\
e_{\texttt{BN-S}} &= \lambda x.\texttt{book}(S); (\lambda y.\texttt{if}\ [y = \texttt{FALSE}]\ \texttt{then}\ \texttt{unbook}(S)\ \texttt{else}\ *) \\
&\qquad ((\texttt{req}_{\rho'}\, Card \xrightarrow{\varphi_{\texttt{BN}}} Bool)x) \\
e_{\texttt{TA}} &= \lambda x.\varphi_{\texttt{TA}}\big[(\texttt{req}_{\bar{\rho}}\, Card \to \mathbf{1})x; (\texttt{req}_{\bar{\rho}'}\, Card \to \mathbf{1})x; rcpt\big]
\end{aligned}
$$

Briefly, $e_{\texttt{PWU}}$ receives a card number x, verifies whether it is a registered one (i.e. $[x \in \mathcal{C}]$) and charges the due amount to x. Service $e_{\texttt{POL}}$ works similarly, but verifies money availability (event `check`) before and after the operation.

Booking services $e_{\texttt{BH-S}}$ require a payment and then, if it has been authorised (i.e. the resource `TRUE` has been received), books the flight (accommodation). On the contrary, $e_{\texttt{BN-S}}$ books the flight (accommodation) and then cancels the reservation if the payment has been refused. Both $e_{\texttt{BH-S}}$ and $e_{\texttt{BN-S}}$ require the behaviour of the invoked service to comply with the policies $\varphi_{\texttt{BH}}$ and $\varphi_{\texttt{BN}}$.

Finally $e_{\texttt{TA}}$ simply calls two booking service instances and releases the receipt $rcpt$ of type Rec to the client. Note that now the travel agency applies its policy $\varphi_{\texttt{TA}}$ to the sequential composition of service requests. □

Operational semantics. Clearly, the run-time behaviour of a network of services depends on the way they interact. As already mentioned, requests do not directly refer to the specific services that will be actually invoked during the execution, but to their abstract behaviour, i.e. to their type and effect (defined below), only. A *plan* resolves the requests by associating them with locations hosting the relevant services. Needless to say, different plans lead to different executions. A plan is said to be *valid* if and only if the executions it drives comply with all the active security policies. Of course, a service network can have many, one or even no valid plans.

A computational step of a program is a transition from a source configuration to a target one. In our model, configurations are pairs η, e where η is the execution *history*, that is the sequence of events done so far (ε being the empty one), and e is the expression under evaluation. Actually, the syntax of histories and expressions is slightly extended with markers $[^m_\varphi$ as explained below in the comment to the rule for framing. The automaton for a policy φ will simply ignore these markers.

Formally, a plan is a (partial) mapping from request identifiers $(\rho, \rho', \ldots)$ to service locations $(\ell, \ell', \ldots)$ defined as

$$\pi, \pi' ::= \emptyset \mid \{\rho \mapsto \ell\} \mid \pi; \pi'$$

An empty plan $\emptyset$ is undefined for any request, while a singleton $\{\rho \mapsto \ell\}$ is only defined for request ρ to be served by the service hosted at location ℓ. Plan composition $\pi; \pi'$ combines two plans. It is defined if and only if for all ρ such that

$$(\mathtt{S{-}Ev_1}) \quad \frac{\eta, e \to_\pi \eta', e'}{\eta, \alpha(e) \to_\pi \eta', \alpha(e')} \qquad (\mathtt{S{-}Ev_2}) \quad \eta, \alpha(r) \to_\pi \eta\alpha(r), *$$

$$(\mathtt{S{-}If}) \quad \eta, \mathtt{if}\ g\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{ff} \to_\pi \eta, e_{\mathcal{B}(g)} \qquad (\mathtt{S{-}App_1}) \quad \frac{\eta, e_1 \to_\pi \eta', e_1'}{\eta, e_1 e_2 \to_\pi \eta', e_1' e_2}$$

$$(\mathtt{S{-}App_3}) \quad \eta, (\lambda_z x.e)v \to_\pi \eta, e\{v/x, \lambda_z x.e/z\} \qquad (\mathtt{S{-}App_2}) \quad \frac{\eta, e_2 \to_\pi \eta', e_2'}{\eta, ve_2 \to_\pi \eta', ve_2'}$$

$$(\mathtt{S{-}Frm_1}) \quad \eta, \varphi[e] \to_\pi \eta[_\varphi^m, \varphi^m[e] \quad m \text{ fresh} \qquad (\mathtt{S{-}Frm_2}) \quad \eta, \varphi^m[v] \to_\pi \eta]_\varphi^m, v$$

$$(\mathtt{S{-}Frm_3}) \quad \frac{\eta[_\varphi^m \eta', e \to_\pi \eta[_\varphi^m \eta'', e' \quad \eta'' \models \varphi}{\eta[_\varphi^m \eta', \varphi^m[e] \to_\pi \eta[_\varphi^m \eta'', \varphi^m[e']}$$

$$(\mathtt{S{-}Req}) \quad \frac{e_{\bar\ell} : \tau \xrightarrow{H} \tau' \in \mathtt{Srv}\rfloor_\ell \qquad \pi(\rho) = \bar\ell \qquad H \models \varphi}{\eta, (\mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau')v \to_\pi \eta, e_{\bar\ell} v}$$

Table 5.10: The operational semantics of λ^{req}.

$\rho \in dom(\pi) \cap dom(\pi') \Rightarrow \pi(\rho) = \pi'(\rho)$, i.e. the same request is never resolved by different services. Two such plans are called *modular*.

Given a plan π we evaluate λ^{req} expressions, i.e. services, according to the rules of the operational semantics given in Table 5.10. Actually, a transition should be also labelled by the location ℓ hosting the expression under evaluation. For readability, we omit this label.

Briefly, an event $\alpha(r)$ is appended to the current history (possibly after the evaluation of its parameter), a conditional branching chooses between two possible executions (depending on its guard g) and application works as usual (recall that v is a value, i.e. either a resource, a variable, an abstraction or a request). The rule ($\mathtt{S{-}Frm_1}$) opens an instance of framing $\varphi[e]$ and records the activation in the history with a marker $[_\varphi^m$, and in the expression with $\varphi^m[e]$ (to keep different instantiations apart we use a fresh m not occurring in the past history η). The rule ($\mathtt{S{-}Frm_2}$) simply deactivates the framing and correspondingly adds the proper marking $]_\varphi^m$ to the history. The rule ($\mathtt{S{-}Frm_3}$) checks whether the history at the right of the m-th instantiation of φ respects this policy; in other words if the history after the activation of that specific instance does not violate φ. Recall that all "opening" and "closing" markers ($[_\varphi^m$ and $]_\varphi^m$) within a history are all distinct and that the usage automata skip them all. This is the way we implement our right-bounded local mechanism.

A service request firstly retrieves the service $e_{\bar\ell}$ that the current plan π associates

with ρ within the repository $\mathtt{Srv}$. We assume that services can not be composed in a circular way. This condition amounts to saying that there exists a partial order relation $\prec$ over services: read $\ell \prec \bar{\ell}$ as ℓ can see $\bar{\ell}$. So the rule says that the selected service must be within the sub-network that can be seen from the client (hosted at location ℓ, the implicit and omitted label of the transition). This is rendered by the check $e_{\bar{\ell}} \in \mathtt{Srv}\rfloor_\ell = \{e_{\ell'} : \tau \in \mathtt{Srv} \mid \ell \prec \ell'\}$. Additionally, the effect of the selected service is checked against the policy φ required by the client. In other words, the client verifies if its requirement φ is met by the "contract" H offered by the service. If successful, the service is finally applied to the value provided by the client, so implementing our "call-by-contract".

We show below a simple execution of the system in our running example.

Example 5.26 Consider the service *Book-Here-F* as implemented in Example 5.25. If it is invoked with parameter $c \in \mathcal{C}$ starting from an empty trace, its execution under the plan $\pi = \{\rho \mapsto \mathtt{PWU}\}$ follows. Note that this computation complies with the only policy $\varphi_{\mathtt{BH}}$ that is activated when the request ρ is evaluated. In the next section we will formalise this notion.

$$
\begin{aligned}
&\varepsilon, (e_{\mathtt{BH-F}}\ c)\\
&\quad \to_\pi \varepsilon, (\lambda_z y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *)((\mathtt{req}_\rho\ Card \xrightarrow{\varphi_{\mathtt{BH}}} Bool)c)\\
&\quad \to_\pi \varepsilon, (\lambda_z y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *)((e_{\mathtt{PWU}})c)\\
&\quad \to_\pi \varepsilon, (\lambda_z y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *)\\
&\qquad\quad (\mathtt{if}\ [c \in \mathcal{C}]\ \mathtt{then}\ \mathtt{charge}(c); \mathtt{TRUE}\ \mathtt{else}\ \mathtt{FALSE})\\
&\to_\pi \varepsilon, (\lambda_z y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *)(\mathtt{charge}(c); \mathtt{TRUE})\\
&\to_\pi \mathtt{charge}(c), (\lambda_z y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *)(\mathtt{TRUE})\\
&\to_\pi \mathtt{charge}(c), \mathtt{if}\ [\mathtt{TRUE} = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(F)\ \mathtt{else}\ *\\
&\to_\pi \mathtt{charge}(c), \mathtt{book}(F)\\
&\to_\pi \mathtt{charge}(c)\mathtt{book}(F), *
\end{aligned}
$$

□

We illustrate below how right-bounded local policies work.

Example 5.27 Let $z = \lambda_z x.\varphi[\alpha(r); z(x)]$ and let φ be the policy saying that "a single $\alpha(r)$ is allowed". Then, we have the following computation (under the empty plan).

$$\varepsilon, (z\ *) \to_\emptyset \varepsilon, \varphi[\alpha(r); z(*)] \to_\emptyset [^1_\varphi, \varphi^1[\alpha(r); z(*)] \to_\emptyset [^1_\varphi \alpha(r), \varphi^1[z(*)]$$

Note that the last transition is possible because $\alpha(r) \models \varphi$. Then the computation proceeds as follows.

$$\to_\emptyset [^1_\varphi \alpha(r), \varphi^1[\varphi[\alpha(r); z(*)]] \to_\emptyset [^1_\varphi \alpha(r) [^2_\varphi, \varphi^1\left[\varphi^2[\alpha(r); z(*)]\right]$$

$H, H' ::=$	ε	empty
	h	variable
	$\alpha(\dot{x})$	access event
	$H \cdot H'$	sequence
	$H + H'$	choice
	$\varphi[H]$	security framing
	$\mu h.H$	recursion
	gH	guard

Table 5.11: The syntax of history expressions.

The last configuration is stuck. Indeed, a further step would lead to the configuration $[^1_\varphi \alpha [^2_\varphi \alpha, \varphi^1[\varphi^2[z(*)]]$. This causes a security violation, in spite of the fact that the second instance of φ is satisfied by the suffix α of the history after the marker $[^2_\varphi$. Indeed, the history after the first instance does not satisfy φ: $\alpha[^2_\varphi \alpha \not\models \varphi$ (recall that the marker $[^2_\varphi$ is invisible to φ). □

5.1.3 Type and effect system

We now introduce our type and effect system for λ^{req}. Our system builds upon [23, 26], aiming at better approximating service behaviour through more precise history expressions. For that, we introduce two new elements: effects with guards and a new typing rule that handles them. A key point is that guards generate invariants that we can exploit for validating the service network even when one or more resources are unspecified.

History expressions We statically check services to comply with given security policies. To do that we soundly over-approximate the behaviour of λ^{req} programs by history expressions, that denote sets of histories. Table 5.11 contains the abstract syntax of history expressions, which extends those of [23] in the explicit treatment of guards.

A history expression can be empty (ε), a single access event to some resource ($\alpha(r)$ or $\alpha(x)$). Also, a history expression can be obtained through sequential composition ($H \cdot H'$) where we assume $H \cdot \varepsilon = \varepsilon \cdot H = H$. History expressions can be combined through non-deterministic choice ($H + H'$) and we feel free to consider "+" associative and to use sometimes the standard abbreviation $\sum_{i \in \{1,\ldots,k\}} H_i$, for a non-deterministic choice among k history expressions. Moreover, we use safety framing $\varphi[H]$ for specifying that all the execution histories represented by H are under the scope of the policy φ. Additionally, $\mu h.H$ (where μ binds the free occurrences

$\alpha(\dot{x}) \xrightarrow{\alpha(\sigma\dot{x})}_\sigma \varepsilon$	$\dfrac{H \xrightarrow{a}_\sigma H'}{H \cdot H'' \xrightarrow{a}_\sigma H' \cdot H''}$	$\dfrac{H \xrightarrow{a}_\sigma \varepsilon}{H \cdot H' \xrightarrow{a}_\sigma H'}$
$\dfrac{H \xrightarrow{a}_\sigma H'}{gH \xrightarrow{a}_\sigma H'} \quad \sigma \models g$	$\dfrac{H \xrightarrow{a}_\sigma H''}{H + H' \xrightarrow{a}_\sigma H''}$	$\dfrac{H' \xrightarrow{a}_\sigma H''}{H + H' \xrightarrow{a}_\sigma H''}$
$\varphi[H] \xrightarrow{[\varphi}_\sigma H\cdot]_\varphi$	$]_\varphi \xrightarrow{]\varphi}_\sigma \varepsilon$	$\dfrac{H\{\mu h.H/h\} \xrightarrow{a}_\sigma H'}{\mu h.H \xrightarrow{a}_\sigma H'}$
$[\![H]\!]^\sigma = \{a_1 \cdots a_n \mid H \xrightarrow{a_1}_\sigma \cdots \xrightarrow{a_n}_\sigma H_n\}$		$n \geqslant 0;\ a_i \in \mathsf{Ev} \cup \{[_\varphi,]_\varphi\}$

Table 5.12: The semantics of history expressions.

of h in H) represents recursive history expressions. Finally, we introduce guarded histories gH (where the guard g is defined according to the syntax of Definition 5.9).

The *operational semantics* function in Table 5.12 maps a history expression H to a set of histories $\mathcal{H}$. Intuitively, the semantics of a history expression H contains all the prefixes of all the traces that H may perform. The traces of a history expression are produced according to the rules of a Labelled Transition System. We write $H \xrightarrow{a}_\sigma H'$ to denote that, under the substitution σ, H performs a and reduces to H'.

We now introduce the semantics of history expressions. Basically, each history expression denotes a set of histories.

The semantics operator $[\![\cdot]\!]^\cdot$ maps a history expression H into a set of histories $\mathcal{H}$, where substitutions are the same as in Section 5.1.2.

As expected, an expression $\alpha(\dot{x})$ can transit to ε while firing the event $\alpha(\sigma(\dot{x}))$ (recall that $\dot{x}$ stands for either a resource r or a variable x). Sequential composition $H \cdot H'$ and non-deterministic choice $H + H'$ behave in the usual way. A policy framing $\varphi[H]$ adds a special event ($[_\varphi$) to denote the policy activation point and appends a corresponding deactivation event, i.e. $]_\varphi$, at the end of the scope of the policy. The history expression gH behaves as H if σ *satisfies* g (in symbols $\sigma \models g$). Satisfiability of a guard g through a substitution σ is equivalent to check whether $\mathcal{B}(\sigma g) = tt$ (remember that $\mathcal{B}$ is total). Finally, $\mu h.H$ behaves as H where all the free instances of h have been replaced by $\mu h.H$.

5.1.4 Typing relation

We define types and type environments in the usual way. Type environments are defined in a standard way as mappings from variables to types. Types can be either

$$\tau, \tau' ::= \mathbf{1} \mid R, R' \subseteq \mathsf{Res} \mid \tau \xrightarrow{H} \tau' \qquad \Gamma, \Gamma' ::= \emptyset \mid \Gamma; x : \tau$$

$$(\texttt{T-Unit})\, \Gamma, \varepsilon \vdash_g * : \mathbf{1} \qquad (\texttt{T-Res})\, \Gamma, \varepsilon \vdash_g r : R \subseteq \mathsf{Res} \qquad (\texttt{T-Var})\, \Gamma, \varepsilon \vdash_g x : \Gamma(x)$$

$$(\texttt{T-Abs})\, \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_g e : \tau'}{\Gamma, \varepsilon \vdash_g \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad (\texttt{T-Ev})\, \frac{\Gamma, H \vdash_g e : R}{\Gamma, H \cdot \sum_{r \in R} \alpha(r) \vdash_g \alpha(e) : \mathbf{1}}$$

$$(\texttt{T-App})\, \frac{\Gamma, H \vdash_g e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_g e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_g e\, e' : \tau'} \qquad (\texttt{T-Frm})\, \frac{\Gamma, H \vdash_g e : \tau}{\Gamma, \varphi[H] \vdash_g \varphi[e] : \tau}$$

$$(\texttt{T-Wkn})\, \frac{\Gamma, H \vdash_g e : \tau \quad \sigma \models g \quad [\![H]\!]^\sigma \subseteq [\![H']\!]^\sigma}{\Gamma, H' \vdash_g e : \tau}$$

$$(\texttt{T-If})\, \frac{\Gamma, H \vdash_{g \wedge g'} e : \tau \quad \Gamma, H \vdash_{g \wedge \neg g'} e' : \tau}{\Gamma, H \vdash_g \texttt{if}\, g' \,\texttt{then}\, e \,\texttt{else}\, e' : \tau} \qquad (\texttt{T-Str})\, \frac{\Gamma, H \vdash_g e : \tau \quad g \Rightarrow g'}{\Gamma, g' H \vdash_g e : \tau}$$

$$(\texttt{T-Req})\, \frac{I = \{H \mid e_{\ell'} : \tau \xrightarrow{H} \tau' \in \texttt{Srv}\rfloor_\ell \wedge H \models \varphi\}}{\Gamma, \varepsilon \vdash_g \texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau' : \tau \xrightarrow{\sum_{H \in I} H} \tau'}$$

Table 5.13: Types, type environment and typing relation.

base types, i.e. unit or resources, or higher-order types $\tau \xrightarrow{H} \tau'$ annotated with the history expression H. Resources belong to subsets $R, R', \ldots$, e.g. *Bool* and *Card* in several examples in this paper. Note that, as expected, service interfaces (see Section 5.1.2) published on the service repository `Srv` are simply higher-order types annotated with a location identifier ℓ.

A typing judgement has the form $\Gamma, H \vdash_g e : \tau$ and means that the expression e is associated with the type τ and the history expression H. The guard g labelling the $\vdash$ records information about the branchings passed through during the typing process. Table 5.13 shows the definitions of types, type environment and the type and effect system.

We require the repository to be trusted — this is the only trust requirement we put on our framework. Consequently, to be included in `Srv`, services must be well-typed with respect to our type and effect system, i.e.

$$e_\ell : \tau \xrightarrow{H} \tau' \in \texttt{Srv} \Longrightarrow \emptyset, \varepsilon \vdash_{tt} e_\ell : \tau \xrightarrow{H} \tau'$$

We briefly comment on the typing rules. We borrow most of them from [23],

except for those dealing with guards, namely (T-If), (T-Str) and (T-Wkn). The rules (T-Unit, T-Res, T-Var) for $*$, resources and variables are straightforward. An event has type **1** and produces a history that is the one obtained from the evaluation of its parameter increased with the event itself (T-Ev). Note that, since the set of resources is finite, an event only has finitely many instantiations.

An abstraction has an empty effect and a functional type carrying a *latent effect*, i.e. the effect that will be produced when the function is actually applied (T-Abs). The application moves the latent effect to the actual history expression and concatenates it with the actual effects according the call-by-value semantics (T-App). Security framing extends the scope of the property φ to the effect of its target (T-Frm).

The rule for conditional branching says that if we can type e and e' to the same τ generating the same effect H, then we can extend τ and H to be the type and effect of the whole expression (T-If). Moreover, in typing the sub-expressions we take into account the guard g' and its negation, respectively. Indeed, the rule requires that the expression e has got type τ under the guard $g \wedge g'$, as e will only be executed if g' is true. Of course also the conditions collected so far in g should be true, which in turn enable the whole $\texttt{if}\, g' \,\texttt{then}\, e \,\texttt{else}\, e'$ to occur at run-time. Symmetrically for e'.

Similarly to abstractions, service requests have an empty effect (T-Req). However, the type of a request is obtained as the composition of all the types of the possible servers. In particular, the resulting latent effect is the (unguarded) non-deterministic choice among them. Note that we only accept exact matching for input/output types. A more general approach would require to define a sub-typing relation. However, such an extension is out of the scope of this work and it is easy, following the proposal of [23], to which we refer the interested reader for details.

The last two rules are for weakening and strengthening. The first (T-Wkn) states that is always possible to make a generalisation of the effect inferred from an expression e, possibly losing precision. In particular, we say that in typing e a history expression H can be replaced by H' provided that H' denotes a superset of the histories denoted by H under any possible environment σ, provided that it satisfies g. Finally, (T-Str) applies a guard g' to an effect H provided that $\forall\, \sigma.\sigma \models g \Rightarrow \sigma \models g'$, abbreviated by $g \Rightarrow g'$. This rule says that we can use the information stored in g for wrapping an effect in a more precise guarded context.

A few examples of typing derivations follow. Recall that resources include credit cards ($\mathcal{C}, \mathcal{C}' \subseteq Card$), receipts ($Rec$) and boolean values ($Bool$).

Example 5.28 Let $e_{\texttt{PWU}}$ be the service introduced in Example 5.25, then the following derivation is possible (dots stands for trivial or symmetrical derivations)

$$\dfrac{\dfrac{\dfrac{\vdots}{x : Card, [x \in \mathcal{C}]\, \texttt{charge}(\mathcal{C}) \vdash_{[x \in \mathcal{C}]} \texttt{charge}(x); \texttt{TRUE} : Bool}\,(\texttt{T-Abs}) \quad \vdots}{x : Card, [x \in \mathcal{C}]\, \texttt{charge}(\mathcal{C}) \vdash_{true} \texttt{IF} : Bool}\,(\texttt{T-If})}{\emptyset, \varepsilon \vdash_{true} e_{\texttt{PWU}} : Card \xrightarrow{[x \in \mathcal{C}]\texttt{charge}(\mathcal{C})} Bool}\,(\texttt{T-Abs})$$

where
$\texttt{charge}(\mathcal{C}) = \sum_{c \in \mathcal{C}} \texttt{charge}(c)$ and $\texttt{IF} = \texttt{if}\ [x \in \mathcal{C}]\ \texttt{then}\, \texttt{charge}(x); \texttt{TRUE}\, \texttt{else}\, \texttt{FALSE}$.

Similarly, we observe that

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{POL}} : Card \xrightarrow{[x \in \mathcal{C}']\texttt{check}(\mathcal{C}')\texttt{charge}(\mathcal{C}')\texttt{check}(\mathcal{C}')} Bool$$

□

Example 5.29 Consider now $e_{\texttt{BH}-\texttt{S}}$ from Example 5.25. For the rightmost expression we can derive

$$(\texttt{T-App})\ \dfrac{x : Card, \varepsilon \vdash_{true} \texttt{req}_{\rho}\, Card \xrightarrow{\varphi_{\texttt{BH}}} Bool : \tau \qquad x : Card, \varepsilon \vdash_{true} x : Card}{x : Card, [x \in \mathcal{C}]\texttt{charge}(\mathcal{C}) \vdash_{true} (\texttt{req}_{\rho}\, Card \xrightarrow{\varphi_{\texttt{BH}}} Bool)x : Bool}$$

where

$$\tau = Card \xrightarrow{[x \in \mathcal{C}]\texttt{charge}(\mathcal{C})} Bool$$

Moreover we can derive

$$\dfrac{\dfrac{\dfrac{\Gamma, \texttt{book}(S) \vdash_{[y=\texttt{TRUE}]} \texttt{book}(S) : \mathbf{1}}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{[y=\texttt{TRUE}]} \texttt{book}(S) : \mathbf{1}}\,(\texttt{T-Str}) \qquad \dfrac{\Gamma, \varepsilon \vdash_{[y \neq \texttt{TRUE}]} * : \mathbf{1}}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{[y \neq \texttt{TRUE}]} * : \mathbf{1}}\,(\texttt{T-Wkn})}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{true} \texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(S)\ \texttt{else}\ * : \mathbf{1}}\,(\texttt{T-If})$$

where $\Gamma = x : Card; y : Bool$. Combining the two we obtain

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{BH}-\texttt{S}} : Card \xrightarrow{[x \in \mathcal{C}]\texttt{charge}(\mathcal{C}) \cdot [y=\texttt{TRUE}]\texttt{book}(S)} \mathbf{1}$$

Similarly, the result of typing $e_{\texttt{BN}-\texttt{S}}$ is

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{BN}-\texttt{S}} : Card \xrightarrow{\texttt{book}(S) \cdot [x \in \mathcal{C}']\texttt{check}(\mathcal{C}')\texttt{charge}(\mathcal{C}')\texttt{check}(\mathcal{C}')[y=\texttt{FALSE}]\texttt{unbook}(S)} \mathbf{1}$$

□

Example 5.30 Consider now the term $e_{\mathtt{TA}}$. The following derivation is possible

$$\dfrac{\dfrac{\dfrac{\vdots}{x : Card, \varepsilon \vdash_{true} \mathtt{req}_{\bar{\rho}}\ Card \to \mathbf{1} : Card \xrightarrow{H+H'} \mathbf{1}}\ (\mathtt{T\text{-}Req}) \qquad \dfrac{}{x : Card, \varepsilon \vdash_{true} x : Card}\ (\mathtt{T\text{-}Var})}{x : Card, H + H' \vdash_{true} (\mathtt{req}_{\bar{\rho}}\ Card \to \mathbf{1})x : \mathbf{1}}\ (\mathtt{T\text{-}App})$$

where

$$H = [x \in \mathcal{C}]\mathtt{charge}(\mathcal{C}) \cdot [y = \mathtt{TRUE}]\mathtt{book}(S)$$

$$H' = \mathtt{book}(S) \cdot [x \in \mathcal{C}']\,\mathtt{check}(\mathcal{C}')\mathtt{charge}(\mathcal{C}')\mathtt{check}(\mathcal{C}') \cdot [y = \mathtt{FALSE}]\mathtt{unbook}(S)$$

Then, we can type the whole service as follows

$$\emptyset, \varepsilon \vdash_{true} e_{\mathtt{TA}} : Card \xrightarrow{\varphi_{\mathtt{TA}}[(H+H')\cdot(H+H')]} Rec$$

□

Our type and effect system produces history expressions that approximate the run-time behaviour of programs. The soundness of our approach relies on producing *safe* history expressions, i.e. any trace produced by the execution of an expression e (under any valid plan) is denoted by the history expression obtained typing e. To prove type and effect safety we need the following lemmata. Lemma 5.1 states that type and effect inference is closed under logical implication for guards, and Lemma 5.2 says that history expressions are preserved by programs execution.

Lemma 5.1 If $\Gamma, H \vdash_g e : \tau$ and $g' \Rightarrow g$ then $\Gamma, H \vdash_{g'} e : \tau$ □

The histories denoted by history expressions are slightly different from those produced at runtime. To compare histories of these two kinds, we introduce below the operator ∂, that removes labels from markers of framing events:

$$\varepsilon^{\partial} = \varepsilon \qquad (\alpha(\dot{x})\eta)^{\partial} = \alpha(\dot{x}) \cdot (\eta^{\partial}) \qquad ([^{m}_{\varphi}\eta)^{\partial} = [_{\varphi}\eta^{\partial} \qquad (]^{m}_{\varphi}\eta)^{\partial} =]_{\varphi}\eta^{\partial}$$

Lemma 5.2 (Subject reduction) Let $\Gamma, H \vdash_g e : \tau$ and $\eta, e \to^*_\pi \eta', e'$. For each g' such that $g' \Rightarrow g$, there exists H' such that $\Gamma, H' \vdash_{g'} e' : \tau$ and $\forall \sigma. \sigma \models g' \Longrightarrow (\eta'[\![H']\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$ □

Now, the soundness of our approach is established by the following theorem, where we overload η to denote both a history generated by the operational semantics of an expression e (i.e. possibly containing framing markers), and a history belonging to the denotational semantics of a history expression H (i.e. without markers).

Theorem 5.1 (Type safety)
If $\Gamma, H \vdash_{true} e : \tau$ and $\varepsilon, e \to^*_\pi \eta', v$, then $\forall\ \sigma.\ \exists\ \eta \in [\![H]\!]^\sigma$ such that $\eta = (\eta')^\partial$. □

We now define the notion of validity for a history expression H. The validity of H guarantees that the expression e, from which H has been derived, will raise no security violations at runtime.

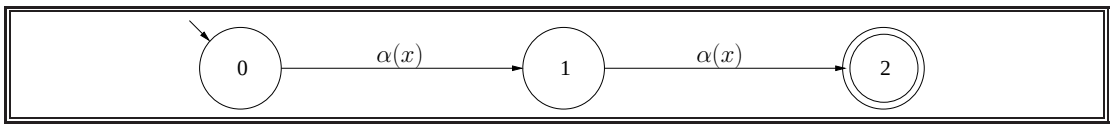


Figure 5.34: A policy saying "never two actions α on the same resource".

Definition 5.7 (Validity of History Expressions)

A history η is *balanced* iff it is produced by the following grammar.

$$B ::= \varepsilon \mid \alpha(\dot{x}) \mid [_\varphi B]_\varphi \mid BB'$$

H is valid iff $\forall\, \sigma$ and $\forall\, \eta[_\varphi \eta']_\varphi \eta'' \in [\![H]\!]^\sigma$ (with η' balanced) then $\eta' \models \varphi$.

The type and effect system of [23] has no rule for strengthening like our rule (`T-Str`). The presence of this rule in our system makes it possible to discard some of the denoted traces. These traces correspond to executions that, due to the actual instantiation of formal parameters, can not take place. Consequently, our type and effect system produces more compact and precise history expressions than those of [23]. This enables the verification algorithm to run faster and to produce fewer false positives, so improving both the efficiency and effectiveness of the original method. Example 5.31 shows that strengthening is crucial in keeping small the size of a history expression H.

As a matter of fact, the validity of H is established through model-checking. Roughly, the model-checking procedure extracts each policy instance from a program. Then, the model-checker verifies whether the set of traces violating the policy and the set of histories denoted by expression in its scope have an empty intersection. If they share no traces, the program cannot violate the policy. This problem is known to be polynomial in the size of the history expression H. We refer the interested reader to [29].

Example 5.31 Consider the policy φ defined by the automaton

$$A_\varphi = \langle \{\alpha\}, \{0,1,2\}, 0, \{2\}, \{\langle 0, \alpha(x), 1\rangle, \langle 1, \alpha(x), 2\rangle\}\rangle$$

depicted in Figure 5.34. Briefly, the automaton recognizes the (illegal) traces containing two α actions on the same resource x. Then consider the program

$$e = \lambda y.\lambda y'.\varphi[\texttt{if}\ [y \neq y']\ \texttt{then}\ \alpha(y); \alpha(y')\ \texttt{else}\ *]$$

We type e in this way:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{y:\tau;y':\tau', H \vdash_{[y\neq y']} \alpha(y);\alpha(y'):\mathbf{1}}\ \text{(T-Str)} \quad \dfrac{\vdots}{y:\tau;y':\tau', H \vdash_{[y=y']} *:\mathbf{1}}\ \text{(T-Wkn)}}{y:\tau;y':\tau', H \vdash_{true} \texttt{if}\ [y\neq y']\ \texttt{then}\ \alpha(y);\alpha(y')\ \texttt{else}\ *:\mathbf{1}}\ \text{(T-If)}}{y:\tau;y':\tau', \varphi[H] \vdash_{true} \varphi[\texttt{if}\ [y\neq y']\ \texttt{then}\ \alpha(y);\alpha(y')\ \texttt{else}\ *]:\mathbf{1}}\ \text{(T-Frm)}}{y:\tau, \varepsilon \vdash_{true} \lambda y'.\varphi[\texttt{if}\ [y\neq y']\ \texttt{then}\ \alpha(y);\alpha(y')\ \texttt{else}\ *]:\tau' \xrightarrow{\varphi[H]} \mathbf{1}}\ \text{(T-Abs)}}{\emptyset, \varepsilon \vdash_{true} e:\tau\to\tau' \xrightarrow{\varphi[H]} \mathbf{1}}\ \text{(T-Abs)}$$

Where $H = [y \neq y']\alpha(y)\alpha(y')$ and the dots represent trivial derivations. In order to verify whether $\varphi[H]$ is valid, we must check if there is some σ such that $[y \neq y']\alpha(y)\alpha(y') \not\models \varphi$. However, we can easily observe that, for any substitution σ, $[\![[y \neq y']\alpha(y)\alpha(y')]\!]^\sigma \subseteq \{\alpha(\sigma y)\alpha(\sigma y') \mid \sigma y \neq \sigma y'\} \cup \{\alpha(\sigma y)\} \cup \{\varepsilon\}$ while $\mathcal{L}(A_\varphi) = \bigcup_\sigma \{\alpha(\sigma x)\alpha(\sigma x)\}$. Hence, the two sets above have an empty intersection and $\varphi[H]$ is valid.

Note that, if we do not apply our strengthening rule when typing e, we can not prove the program to be policy-compliant, as it happens with the proposal in [23, 26] that in this case results in a false positive. □

5.2 Modular plans for secure service composition

In this section we introduce our mechanism for *plan composition*. Roughly, we define a composition strategy allowing for synthesizing a global orchestration plan. The main issue here is defining the necessary conditions under which partial plans can be safely composed. The result is a plan that composes the network services in a secure manner.

5.2.1 Properties of Plans

In Section 5.1.2 we saw how plans drive the execution of services turning service requests into actual service invocations. We can also interpret plans as static descriptions of the dynamic interactions among services. In other words, a plan can be used to replace services requests with the code of the service to which the request is mapped. Note that, applying this substitution process is always terminating since the binding between a request and a service in statically defined. We call this semantics preserving transformation *flattening*, and *flat* a service without requests.

The definition is given in Table 5.14. Intuitively, flattening an expression e with increasingly more defined plans (i.e. plans containing more binding) reduces the set of histories associated with e. This property is made precise below.

Theorem 5.2 Given two plans π and π', if $\Gamma, H \vdash_g e \mid_\pi : \tau$ and $\Gamma, H' \vdash_g e \mid_{\pi;\pi'} : \tau$ then $\forall\, \sigma. [\![H']\!]^\sigma \subseteq [\![H]\!]^\sigma$ □

$$
\begin{array}{ccc}
* \mid_\pi = * & r \mid_\pi = r & x \mid_\pi = x \\
\alpha(e) \mid_\pi = \alpha(e \mid_\pi) & (\lambda_z x.e) \mid_\pi = \lambda_z x.e \mid_\pi & (e\, e') \mid_\pi = e \mid_\pi \ e' \mid_\pi \\
\varphi[e] \mid_\pi = \varphi[e \mid_\pi] & (\texttt{if}\ g\ \texttt{then}\ e\ \texttt{else}\ e') \mid_\pi = \texttt{if}\ g\ \texttt{then}\ e \mid_\pi\ \texttt{else}\ e' \mid_\pi & \\
(\texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau') \mid_\pi = \begin{cases} e_\ell \mid_\pi & \text{if } \pi(\rho) = \ell \wedge e_\ell : \tau \xrightarrow{H_\ell} \tau' \in \texttt{Srv} \wedge H_\ell \models \varphi \\ \texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau' & \text{otherwise} \end{cases} &&
\end{array}
$$

Table 5.14: The flattening operator.

The following definition introduces the key concept of plan *completeness*. Intuitively, a plan π is complete with respect to a service e if it is valid and makes e flat. Complete plans play a crucial role in the secure composition of services.

Definition 5.8

Given a closed term e, a plan π is said to be *complete* for e if and only if

1. $e \mid_\pi$ is flat and
2. for all H such that $\emptyset, H \vdash_{true} e \mid_\pi : \tau$, it is the case that H is valid.

Note that, according to the previous definition, we check the completeness of plans by typing flattened services. However, this method is not suitable for real scenarios. Indeed, typing a service is usually a cumbersome operation that should be applied only if necessary. Moreover, in general we cannot assume the code of the services to be publicly accessible to external entities, e.g. an orchestrator. Hence, implementing this approach could be infeasible.

Nevertheless, more efficient techniques for checking plans exist. For instance, it is possible to extend the syntax of the history expressions and the rules of the type and effect system. This method requires history expressions to be translated in a normal form (i.e. regularised and linearised) and it also need a more complex model-checker for verifying the validity of the history expressions. We refer the interested reader to [23].

However, more complex and efficient approaches do not affect the following dissertation. Indeed, here we aim at defining a bottom-up strategy for composing plans. Such a result is independent from which method is used for the verification of plans, we only require it to be effective.

Complete, modular plans for services can be composed preserving completeness, as stated below.

Lemma 5.3 Given two terms e,e' and two modular plans π,π' complete for e and e', respectively, then $\pi;\pi'$ is complete for both e and e'. □

We now state a theorem characterising our notion of composition. Unfortunately, we cannot aim at a full compositionality, because history validity itself is not compositional. The reason why is that we follow a history-dependent approach to security. The following example suffices to make this point. Let φ say "never $\alpha(r)$ twice". Clearly, $\alpha(r)$ complies with φ, but $\alpha(r)\alpha(r)$ does not.

Lemma 5.3 above helps in finding conditions that permit to obtain secure services by putting together components already proved secure. We shall then outline a way of efficiently reuse the proofs of validity already known for components, so to incrementally prove the validity of a plan for the composed service. (Recall that services are closed expressions, typable in an empty environment.)

Theorem 5.3 Let π_i be modular plans complete for services e_i, and let $\emptyset, H_i \vdash_{g_i} e_i : \tau_i$, $i = 0, 1$. Then,

- $\pi_0; \pi_1$ is complete for both $\alpha(e_0)$ and $\texttt{if}\, g\, \texttt{then}\, e_0\, \texttt{else}\, e_1$
- $\forall \varphi$, if $H_0 \models \varphi$ then
  - $\pi_0; \pi_1$ is complete for $\varphi[e_0]$
  - $\{\rho \mapsto \ell\}; \pi_0$ is complete for $\texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$, with $e_\ell = e_0$ and $e_\ell : \tau \xrightarrow{H_0} \tau' \in \texttt{Srv}$
- if $\tau_0 = \tau_1 \xrightarrow{H_0'} \tau'$ and $H_0 \cdot H_1 \cdot H_0'$ is valid, then $\pi_0; \pi_1$ is complete for $e_0 e_1$

□

Example 5.32 Consider again the service network of introduced in Section 5.1.1. We typed the services of the network in Section 5.1.3 and we defined their security policies in Figure 5.33.

It is immediate to verify that the plans $\pi_1 = \{\rho \mapsto \texttt{PWU}\}$ and $\pi_2 = \{\rho' \mapsto \texttt{POL}\}$ are modular and complete for Book-Here-F (Book-Here-H) and Book-Now-H (Book-Now-F), respectively (actually they are such for the expressions $e_{\texttt{BH-F}}$, $e_{\texttt{BH-H}}$, $e_{\texttt{BN-H}}$ and $e_{\texttt{BN-F}}$).

We now obtain a plan for the service Travel Agency by composing the above two plans (recall that sequentialization is encodable as a simple form of application). To find a valid plan we have to check the validity of the history expressions arising when mapping the requests of $e_{\texttt{TA}}$ into actual servers. The number of composition plans for the small network rooted in the Travel Agency service is 2^8 (2^4 pairs for the Travel Agency and 2^4 mappings for the booking services) However, we already discovered that only one plan, i.e. $\pi_1; \pi_2$, is viable for the booking services. Hence, we can focus on finding an extension of $\pi_1; \pi_2$ also satisfying $\varphi_{\texttt{TA}}$. This consists of verifying only 2^4 history expressions, i.e. those arising from the possible mappings of the two requests $\bar{\rho}$ and $\bar{\rho}'$ (see Example 5.25) into the four booking services.

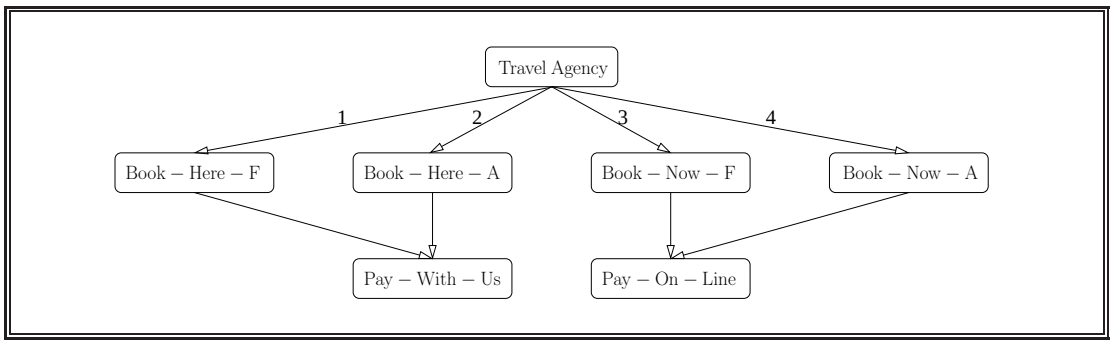


Figure 5.35: Safe plans for the travel booking network

We can prove the following eight plans to be safe:

$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BH} - \texttt{F}, \bar{\rho}' \mapsto \texttt{BH} - \texttt{H}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BH} - \texttt{F}, \bar{\rho}' \mapsto \texttt{BN} - \texttt{H}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BN} - \texttt{F}, \bar{\rho}' \mapsto \texttt{BH} - \texttt{H}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BN} - \texttt{F}, \bar{\rho}' \mapsto \texttt{BN} - \texttt{H}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BH} - \texttt{H}, \bar{\rho}' \mapsto \texttt{BH} - \texttt{F}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BH} - \texttt{H}, \bar{\rho}' \mapsto \texttt{BN} - \texttt{F}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BN} - \texttt{H}, \bar{\rho}' \mapsto \texttt{BH} - \texttt{F}\}$$
$$\{\rho \mapsto \texttt{PWD}, \rho' \mapsto \texttt{POL}, \bar{\rho} \mapsto \texttt{BN} - \texttt{H}, \bar{\rho}' \mapsto \texttt{BN} - \texttt{F}\}$$

As a matter of fact, $\varphi_{\texttt{TA}}$ requires "never `book` twice the same resource" (see Fig. 5.33c). However, each booking service performs at most one `book` action on a single resource. Hence, we are guaranteed that invoking two booking services acting on different resources $\varphi_{\texttt{TA}}$ cannot be violated.

Figure 5.35 shows the eight valid composition plans. Unlabelled arrows denote fixed mapping (shared by every plan). Numbered arrows represents the mappings for the two request of Travel Agency. Odd arrows point to flight booking services, while even arrows point to hotel booking services. Accordingly to what we said above, valid plans corresponds to a pair of these arrows where the first is odd and the second is even or vice versa.

In conclusion, starting from a space of 2^8 possible composition plans for the network rooted in Travel Agency, we discharged many, potentially unsafe plans and we detected few of them (8) that can be proved valid.

□

5.2.2 The "Buy Something" Case Study

Hereafter we present our working example which models the W3C "buy something" scenario for web services [187]. We take into account the web service templates

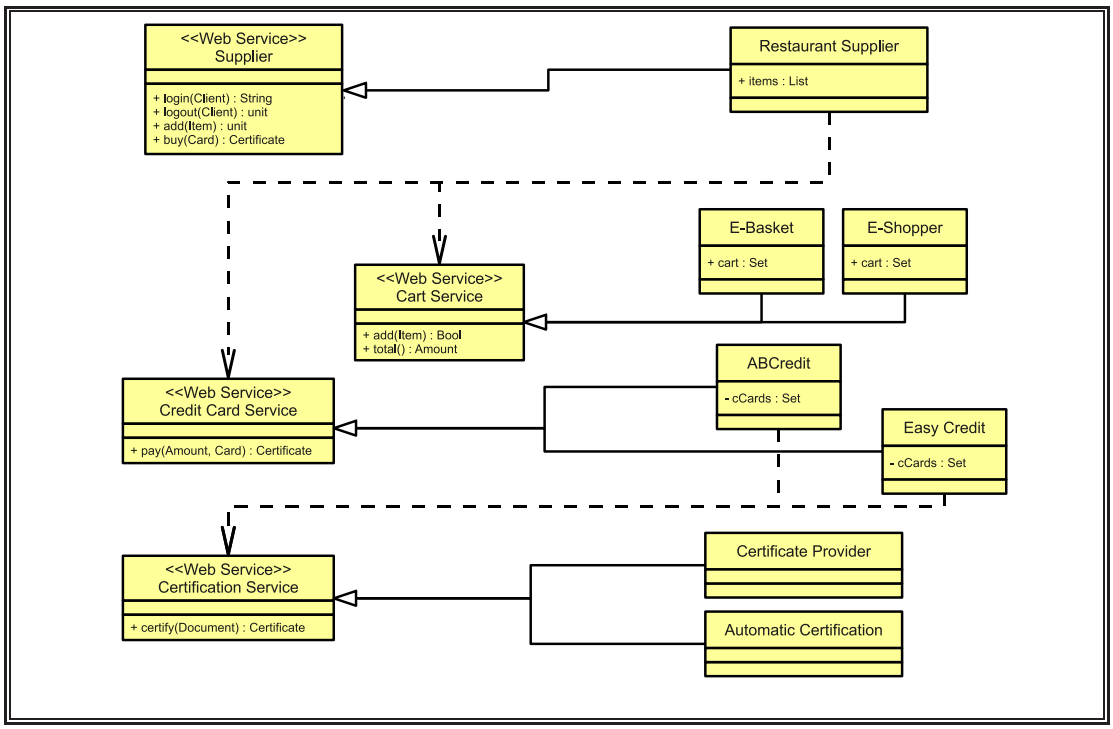


Figure 5.36: The "buy something" scenario.

of the original "buy something" specification and we imagine a simple network implementing it. Then we add to the actual services some security requirement that the network should respect at runtime.

The UML class diagram in Figure 5.36 shows the entities composing the network and their relationships. On the left hand side, web services, denoted by the `<<Web Service>>` stereotype, are specified through a functional interface, that is a specification of the service operations given through signatures.

The right side shows instances of services. Solid lines link a service instance to the interface it implements. Instances can use internal resources for managing their state. When an actual service implements a service interface it has to provide an implementation, e.g. a method or a procedure, for each of the declared operations.

Dashed lines denote service composition relations, i.e. they connect a service instance to a service interface whenever one or more of the source operations implementations invoke (an operation of) the pointed service. Note that these relationships are not defined by the web service interface. Indeed, each service instance implementing a given interface must simply guarantee the interface functionalities.

The service interfaces involved in this case study are: *Supplier*, *Cart Service*, *Credit Card Service* and *Certification Service*. We briefly comment on them.

A *Supplier* is a service offering operations for web-based selling services. These operations are:

- *login*: a client provides its identity (*Client*), receives back a welcome message (of type *String*) and starts a new session.
- *logout*: a client provides its identity (*Client*) and closes its current session.
- *add*: a client adds an item (*Item*) to its current set of items.
- *buy*: a client uses its credit card (*Card*) to buy the set of items that have been added during the session and receives back a certificated receipt (of type *Certificate*).

A *Cart Service* provides the basic support for handling electronic baskets. Its operations are:

- *add*: an item (*Item*) is added to the cart and a boolean acknowledgement is returned.
- *total*: the total value (*Amount*) of the current cart content is returned.

The *Credit Card Service* offers a single, on-line payment operation returning a certified receipt (having type *Certificate*) whenever a certain amount (*Amount*) is charged on the client credit card (*Card*). Finally, the *Certification Service* simply takes a document (*Document*) and returns a signed version of it (*Certificate*).

For each of these interfaces the network contains at least one instance. The *Restaurant Supplier* is the only implementation of the *Supplier* class. Intuitively, it offers a public catalogue of items, stored in a public list. Moreover, its methods invoke other services as part of their implementation. In particular, the *Restaurant Supplier* relies on a *Cart Service* to handle the shopping sessions details of its clients and a *Credit Card Service* to perform payments. Both *E-Basket* and *E-Shopper* implements of the *Cart Service* specification. They use a public resource, i.e. *cart*, and they refer to no external service in their implementation. Two alternative implementations for the *Credit Card Service* exist in the network: *ABCredit* and *Easy Credit*. These two instances have a private, static resource called *cClient* containing the set of the known clients. Moreover, these instances both require a *Certification Service* to sign their receipts. Finally, two instances of the *Certification Service* are in the network.

Policies and network structure. We imagine that each service defines its own security requirements using local policies. Clearly, different service instances can focus on different security aspects and can require customised security policies. For instance, *Restaurant Supplier* can be interested in preventing the undesired usage of its clients' personal data. Hence, it can apply a policy saying "never store details of credit cards" on the invoked services. Similarly, the instances of the *Credit Card Service* can specify requirements on the *Certifying Service* behaviour. Examples of

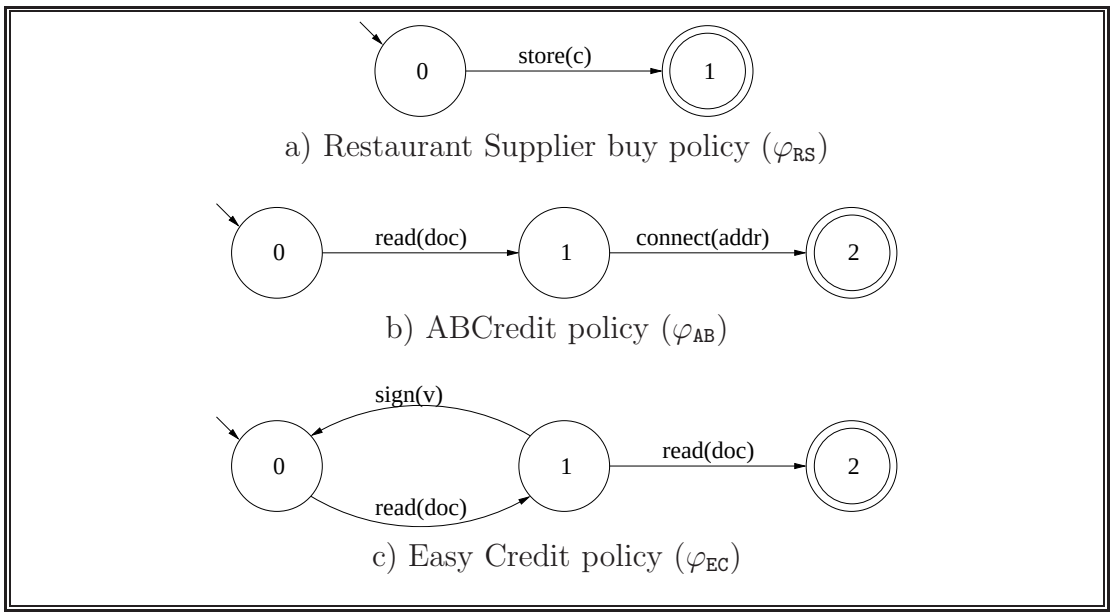


Figure 5.37: Usage automata.

policies of this kind are "never open connections after reading the target document" or "do not read more than what is actually signed".

The usage automata in Figure 5.37 correspond to the properties informally described above. For the sake of readability we use here the graphical notation to define usage automata. Nevertheless, usage automata can be defined within the service implementation or specification, for instance using the approach shown in Section 3.1.

Finally, Figure 5.38 shows the deployment of web services on physical platforms. Again, we assume a simple deployment strategy in which the methods provided by a single instance of a service are place on the same hosting location. However, our model also admits distributed deployment strategies. Indeed, the only requirement that we make on the deployment of services is that there exists no ambiguity between their locations. In this example, all the locations are identified by a public address and the functionalities installed on them are associated to a local port. In this way, each method is uniquely identified inside the network by a pair of coordinates, i.e. the address of its host and its local port.

The deployment diagram has been enriched with relationships, denoted by solid lines, representing all the possible service compositions. A line connects a client, on the left, to a server, on the right, when they are compatible according to the specification of Figure 5.36. Clearly, only one server will be chosen at runtime. Thus, many of the represented connections will not arise during a single execution of this network.

This network is intrinsically incomplete because there are no clients for the

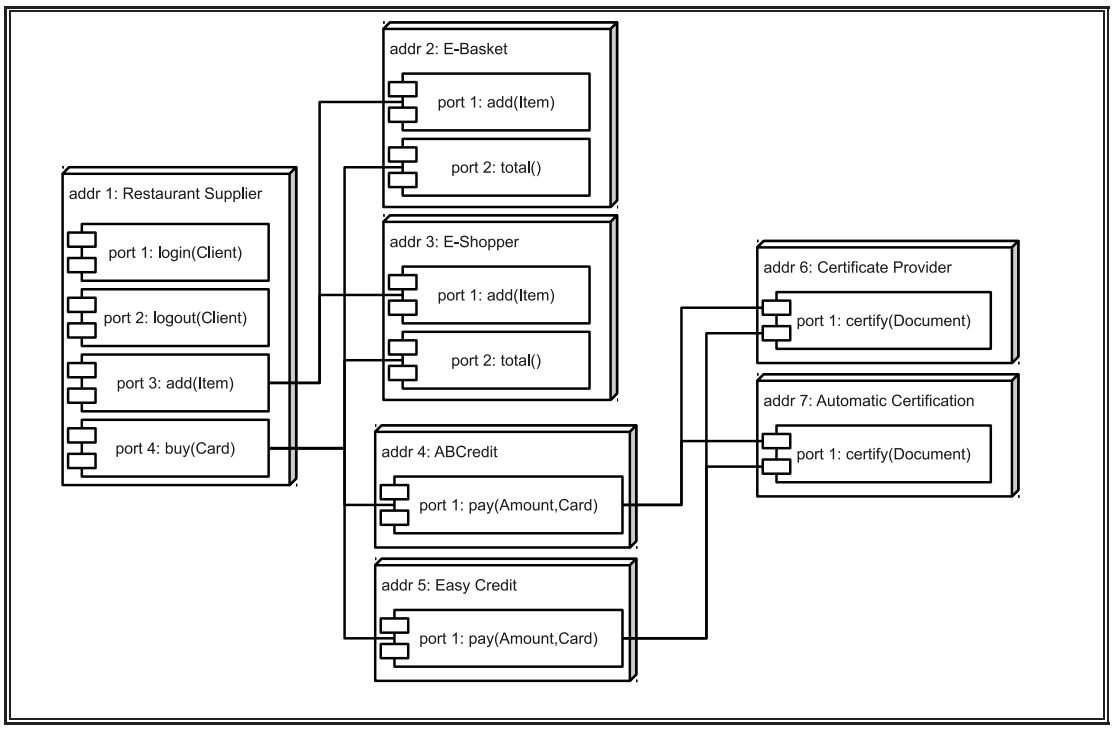


Figure 5.38: Services deployment.

Restaurant Supplier service. However, the mentioned security policies do not involve any aspect of the possible clients behaviour. Indeed, all the services only define security constraints over their own traces and on those of the services they invoke. In particular, each service could be interested in verifying whether the interaction with others satisfies the existing policies or not.

Services implementation. Here we propose an implementation of some of the functions involved in the "buy something" case study. We start by introducing some abbreviations that we use in order to make our presentation more compact and readable.

$$\texttt{let}\, x = e\, \texttt{in}\, e' \;=\; (\lambda x.e')\, e$$

$$\langle v, v' \rangle \;=\; \lambda f.(f v) v'$$

$$\texttt{fst} \;=\; \lambda x.\lambda y.x$$

$$\texttt{snd} \;=\; \lambda x.\lambda y.y$$

$$\tau \times \tau' = (\tau \to (\tau' \to \tau'')) \to \tau'' \qquad \text{for some } \tau''$$

According to the deployment diagram in Figure 5.38 we give an implementation of the functions *buy*, *pay* and *certify* as shown in Table 5.15. In words, the services of Table 5.15 behave as follows.

$$
\begin{array}{lcl}
e_{(1,4)} & = & \lambda x.\texttt{let}\, a = (\texttt{req}_{\rho_1}\, \mathbf{1} \to \mathit{Amount}) * \texttt{in} \\
& & \quad (\texttt{req}_{\rho'_1}\, \mathit{Amount} \times \mathit{Card} \xrightarrow{\varphi_{\texttt{RS}}} \mathit{Certificate})\langle a, x\rangle \\
e_{(4,1)} & = & \lambda x.\texttt{let}\, a = (\texttt{fst}\ x)\, \texttt{in}\, \texttt{let}\, c = (\texttt{snd}\ x)\, \texttt{in} \\
& & \quad \texttt{if}\ [c \in \texttt{cCards}] \\
& & \quad\quad \texttt{then}\ \texttt{access}(c); \texttt{charge}(a); (\texttt{req}_{\rho_4}\, \mathit{Document} \xrightarrow{\varphi_{\texttt{AB}}} \mathit{Certificate})\mathit{rcpt} \\
& & \quad\quad \texttt{else}\ \texttt{store}(c); (\texttt{req}_{\rho'_4}\, \mathit{Document} \to \mathit{Certificate})\mathit{empty} \\
e_{(5,1)} & = & \lambda x.\texttt{let}\, a = (\texttt{fst}\ x)\, \texttt{in}\, \texttt{let}\, c = (\texttt{snd}\ x)\, \texttt{in} \\
& & \quad \texttt{if}\ [c \in \texttt{cCards}] \\
& & \quad\quad \texttt{then}\ \texttt{access}(c); \texttt{charge}(a); (\texttt{req}_{\rho_5}\, \mathit{Document} \xrightarrow{\varphi_{\texttt{EC}}} \mathit{Certificate})\mathit{rcpt} \\
& & \quad\quad \texttt{else}\ (\texttt{req}_{\rho'_5}\, \mathit{Document} \to \mathit{Certificate})\mathit{empty} \\
e_{(6,1)} & = & \lambda_z x.\texttt{let}\, c = \texttt{if}\ [x = \mathit{empty}]\, \texttt{then}\, \mathit{cert}\, \texttt{else}\, \texttt{read}(x); \texttt{sign}(*); z\, x\, \texttt{in} \\
& & \quad \texttt{connect}(\mathit{host}); c \\
e_{(7,1)} & = & \lambda_z x.\texttt{if}\ [x = \mathit{empty}]\, \texttt{then}\, \mathit{cert}\, \texttt{else} \\
& & \quad \texttt{read}(x); \texttt{let}\, c = z\, x\, \texttt{in}\, \texttt{sign}(*); c
\end{array}
$$

Table 5.15: A possible implementation of *buy*, *pay* and *certify*.

- $e_{(1,4)}$ implements the function *buy*. It receives a credit card, denoted by the variable x, retrieves an amount to pay, through a proper invocation, stores the obtained value in the variable a and performs a payment request using the pair $\langle a, x\rangle$. Note that the first request has no policy label. This means that no security constraints are put on the actually invoked service. Instead, the second request carries on a reference to the policy φ_{RS} (see Figure 5.37).

- $e_{(4,1)}$ and $e_{(5,1)}$ both implement the method *pay*. The both of them check whether the received credit card number, associated to x, belongs to a known client ($[c \in \texttt{cCards}]$). If it is the case, they access the card details, charge the payment amount and certify the customer receipt ($rcpt$) through a proper invocation to a certifying service. Otherwise, they return an empty certificate. The two instances apply their own security policies ($\varphi_{\texttt{AB}}$ and $\varphi_{\texttt{EC}}$) on the certification requests whenever they involves a non-empty receipt retrieval. The main difference between the two instances is that $e_{(4,1)}$ stores the credit card number before producing the empty certificate while $e_{(5,1)}$ does not.

- $e_{(6,1)}$ and $e_{(7,1)}$ implement the operation *certify*. The former recursively reads and signs the document x until the end is reached ($[x = \mathit{empty}]$). Then, when the process terminates, $e_{(6,1)}$ opens a connection with *host* and returns the produced certificate c. Instead, $e_{(7,1)}$ first reads the whole document x and then performs all the signing actions before returning c.

We use the following example to show an instance of computation for the previ-

ously presented services.

Example 5.33 Consider the implementation of *pay* given through $e_{(4,1)}$. We simulate the invocation of the function *pay* with actual parameter $\langle \bar{a}, \bar{c} \rangle$, where $\bar{c} \notin \texttt{cCards}$. The execution takes place under a plan π mapping ρ'_4 to $(6,1)$. Starting from the empty trace, the resulting execution is:

$$
\begin{array}{l}
\varepsilon, (e_{(4,1)} \langle \bar{a}, \bar{c} \rangle) \\
\rightarrow_\pi \varepsilon, \texttt{let}\ a = (\texttt{fst}\ \langle \bar{a}, \bar{c} \rangle)\ \texttt{in}\ \texttt{let}\ c = (\texttt{snd}\ \langle \bar{a}, \bar{c} \rangle)\ \texttt{in}\ \texttt{if}\ [c \in \texttt{cCards}] \\
\quad \texttt{then}\ \texttt{access}(c); \texttt{charge}(a); (\texttt{req}_{\rho_4}\ Document \xrightarrow{\varphi_{\texttt{AB}}} Certificate) rcpt \\
\quad \texttt{else}\ \texttt{store}(c); (\texttt{req}_{\rho'_4}\ Document \rightarrow Certificate) empty \\
\rightarrow^*_\pi \varepsilon, \texttt{let}\ c = (\texttt{snd}\ \langle \bar{a}, \bar{c} \rangle)\ \texttt{in}\ \texttt{if}\ [c \in \texttt{cCards}] \\
\quad \texttt{then}\ \texttt{access}(c); \texttt{charge}(\bar{a}); (\texttt{req}_{\rho_4}\ Document \xrightarrow{\varphi_{\texttt{AB}}} Certificate) rcpt \\
\quad \texttt{else}\ \texttt{store}(c); (\texttt{req}_{\rho'_4}\ Document \rightarrow Certificate) empty \\
\rightarrow^*_\pi \varepsilon, \texttt{if}\ [\bar{c} \in \texttt{cCards}] \\
\quad \texttt{then}\ \texttt{access}(\bar{c}); \texttt{charge}(\bar{a}); (\texttt{req}_{\rho_4}\ Document \xrightarrow{\varphi_{\texttt{AB}}} Certificate) rcpt \\
\quad \texttt{else}\ \texttt{store}(\bar{c}); (\texttt{req}_{\rho'_4}\ Document \rightarrow Certificate) empty \\
\rightarrow_\pi \varepsilon, \texttt{store}(\bar{c}); (\texttt{req}_{\rho'_4}\ Document \rightarrow Certificate) empty \\
\rightarrow_\pi \texttt{store}(\bar{c}), (\texttt{req}_{\rho'_4}\ Document \rightarrow Certificate) empty \\
\rightarrow_\pi \texttt{store}(\bar{c}), (e_{(6,1)}) empty \\
\rightarrow^*_\pi \texttt{store}(\bar{c}), \texttt{let}\ c = cert\ \texttt{in}\ \texttt{connect}(host); c \\
\rightarrow_\pi \texttt{store}(\bar{c}), \texttt{connect}(host); cert \\
\rightarrow_\pi \texttt{store}(\bar{c}) \texttt{connect}(host), cert
\end{array}
$$

Checking plans validity. Here we show how to build a valid plan for (a part of) the network depicted in Figure 5.38. We start by typing the implementations of the services in Table 5.15. Below we show the reasoning applied during the typing process.

Let consider the service $e_{(6,1)}$. The following derivations are possible (dots stands for trivial or symmetrical derivations).

$$
(\texttt{T}-\texttt{Abs}) \quad \frac{\begin{array}{c}\vdots \\ \Gamma; c : \tau', \texttt{connect}(host) \vdash_{true} \texttt{connect}(host); c : \tau'\end{array}}{\Gamma, \varepsilon \vdash_{true} \lambda c.\texttt{connect}(host); c : \tau' \xrightarrow{\texttt{connect}(host)} \tau'}
$$

where

$$\Gamma = z : \tau \xrightarrow{H} \tau'; x : \tau$$

$$\tau = Document$$

$$\tau' = Certificate$$

and

$$H = \mu h.[x \neq empty]\texttt{read}(\mathcal{D})\texttt{sign}(*) \cdot h$$

Similarly, we observe that

$$\dfrac{\dfrac{\Gamma, \varepsilon \vdash_g cert : \tau' \quad [\![\varepsilon]\!]^\sigma \subseteq [\![H]\!]^\sigma}{\Gamma, H \vdash_g cert : \tau'} \quad \dfrac{\dfrac{\vdots}{\Gamma, H' \vdash_{\neg g} \texttt{read}(x); \texttt{sign}(*); z\, x : \tau' \quad [\![H']\!]^\sigma \subseteq [\![H]\!]^\sigma}}{\Gamma, H \vdash_{\neg g} \texttt{read}(x); \texttt{sign}(*); z\, x : \tau'}}{\Gamma, H \vdash_{true} \texttt{if}\ [x = empty]\ \texttt{then}\ cert\ \texttt{else}\ \texttt{read}(x); \texttt{sign}(*); z\, x : \tau'}$$

(T−Wkn) (T−If)

where $g = [x = empty]$ and $H' = [x \neq empty]\texttt{read}(\mathcal{D})\texttt{sign}(*) \cdot H$.
Composing the two previous derivations we obtain

$$\emptyset, \varepsilon \vdash_{true} e_{(6,1)} : Document \xrightarrow{H_{(6,1)}} Certificate$$

where $H_{(6,1)} = (\mu h.[x \neq empty]\texttt{read}(\mathcal{D})\texttt{sign}(*) \cdot h) \cdot \texttt{connect}(host)$.

A similar reasoning procedure can be applied for typing $e_{(7,1)}$ to obtain

$$\emptyset, \varepsilon \vdash_{true} Document \xrightarrow{H_{(7,1)}} Certificate$$

where $H_{(7,1)} = \mu h.[x \neq empty]\texttt{read}(\mathcal{D}) \cdot h \cdot \texttt{sign}(*)$

Consider now $e_{(4,1)}$ and $e_{(5,1)}$. We type them in the following way.

$$\emptyset, \varepsilon \vdash_{true} e_{(4,1)} : Amount \times Card \xrightarrow{H_{(4,1)}} Certificate$$

$$\emptyset, \varepsilon \vdash_{true} e_{(5,1)} : Amount \times Card \xrightarrow{H_{(5,1)}} Certificate$$

where:
$H_{(4,1)} = [c \in \texttt{cCards}]\texttt{access}(c)\texttt{charge}(a)H_{(7,1)} + \neg[c \in \texttt{cCards}]\texttt{store}(c) \cdot (H_{(6,1)} + H_{(7,1)})$
and
$H_{(5,1)} = [c \in \texttt{cCards}]\texttt{access}(c)\texttt{charge}(a)H_{(6,1)} + \neg[c \in \texttt{cCards}](H_{(6,1)} + H_{(7,1)})$

Let now apply the typing rules to $e_{(1,4)}$. We obtain the following result:

$$\emptyset, \varepsilon \vdash_{true} e_{(1,4)} : Card \xrightarrow{H_{(1,4)}} Certificate$$

where $H_{(1,4)} = (H_{(2,2)} + H_{(3,2)}) \cdot H_{(5,1)}$.

The typing process presented above is exploited for generating the history expressions that we use for checking the validity of composition plans. We proceed bottom-up as follows. The bottom-level services are implemented by $e_{(6,1)}$ and $e_{(7,1)}$ and they both give rise to the empty plan.

Now, we can build a plan for $e_{(4,1)}$. Its request ρ_4 can only be served by $e_{(7,1)}$. Indeed, it is simple to verify the $H_{(7,1)} \models \varphi_{\texttt{AB}}$ (note that $H_{(7,1)}$ does not even contain the event `connect`). Instead, $H_{(6,1)} \not\models \varphi_{\texttt{AB}}$. Request ρ'_4 can instead be served by both $e_{(6,1)}$ and $e_{(7,1)}$. Collecting the above, we obtain two partial plans:
$\pi_1 = \{\rho_4 \mapsto (7,1), \rho'_4 \mapsto (6,1)\}$ and $\pi'_1 = \{\rho_4 \mapsto (7,1), \rho'_4 \mapsto (7,1)\}$.

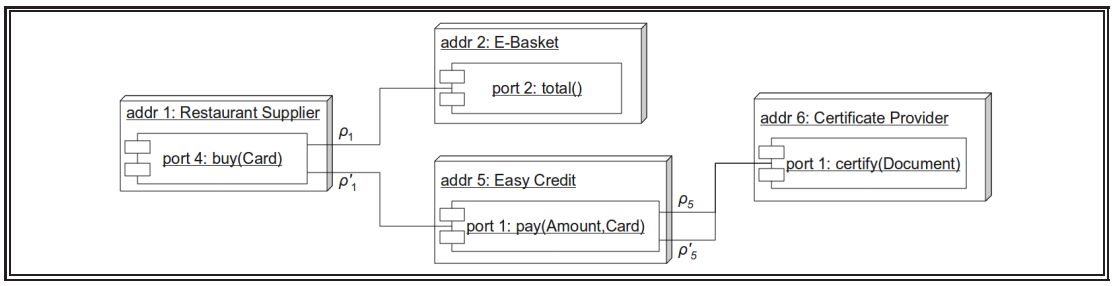


Figure 5.39: A valid plan rooted in service *buy*.

Similarly, for $e_{(5,1)}$ we obtain the two plans:

$\pi_2 = \{\rho_5 \mapsto (6,1), \rho'_5 \mapsto (6,1)\}$ and $\pi'_2 = \{\rho_5 \mapsto (6,1), \rho'_5 \mapsto (7,1)\}$.

We can now show how we get the plan for the topmost service *buy* implemented by $e_{(1,4)}$.

Typing $e_{(1,4)}$ we observe that the request ρ_1 can be served by either the expression $e_{(2,2)}$, i.e. E-Basket, or $e_{(3,2)}$, i.e. E-Shopper. This is recorded in the derivation by the history expression $H_{(2,2)} + H_{(3,2)}$. Since $\mathtt{req}_{\rho_1}$ requires no security policy, we can generate the two alternative (fragments of) valid plans. Similarly, in a valid plan $\mathtt{req}_{\rho'_1}$ will be associated with $e_{(5,1)}$ only. Indeed, $H_{(5,1)} \models \varphi_{\mathtt{RS}}$, while $H_{(4,1)} \not\models \varphi_{\mathtt{RS}}$ as it may perform an event `store`.

As seen above, ρ_1 can be served by either $(2,2)$ or $(3,2)$, while ρ'_1 by $(5,1)$, only. The two resulting partial plans are:

$\pi = \{\rho_1 \mapsto (2,2), \rho'_1 \mapsto (5,1)\}$ and $\pi' = \{\rho_1 \mapsto (3,2), \rho'_1 \mapsto (5,1)\}$.

Composing the previously introduced valid plans according to the rules stated in Theorem 5.3, we obtain the following complete plans.

$$\{\rho_1 \mapsto (2,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (6,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (6,1)\}$$
$$\{\rho_1 \mapsto (2,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (6,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (7,1)\}$$
$$\{\rho_1 \mapsto (2,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (7,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (6,1)\}$$
$$\{\rho_1 \mapsto (2,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (7,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (7,1)\}$$
$$\{\rho_1 \mapsto (3,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (6,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (6,1)\}$$
$$\{\rho_1 \mapsto (3,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (6,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (7,1)\}$$
$$\{\rho_1 \mapsto (3,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (7,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (6,1)\}$$
$$\{\rho_1 \mapsto (3,2), \rho'_1 \mapsto (5,1), \rho_4 \mapsto (7,1), \rho'_4 \mapsto (7,1), \rho_5 \mapsto (6,1), \rho'_5 \mapsto (7,1)\}$$

All these plans are valid and drive executions of the service *buy* raising no security exception. We show in Figure 5.39 the effect of applying the first of the above plan to the service *buy*. Solid lines represent the actual bindings between a client (on the left side) and a server (on the right). Each binding is labelled with the identifier of the request it maps to a service location.

5.3 Synthesizing security prerequisites

The model presented so far follows the standard call-by-contract philosophy, in which service invocation can happen whenever a certain service instance offers a contract that satisfies the client requirements. This technique assumes that a service is not concerned about discriminating over requests, possibly deciding to reject a client. Indeed, in many practical cases, a service aims at accepting as many clients as possible. However, it frequently happens that a service requires some information about its clients, i.e. a prerequisite, before serving them.

We saw in the previous sections how the contract-driven generation of valid plans can be used to organise groups of services within a large scale network. History expressions have been exploited for defining behavioural contracts that are inferred from the implementation of the services. Hyper-local policies (introduced in Section 5.1.1) are applied by the service developers to their code and are automatically included in the history expressions. In this way, a service which is only aware about a small part of the global network it must operate into, can still find a suitable composition. Even though these assumptions seem to be reasonable for many practical cases, one could be interested in extending the security requirements to the composition with the clients of a service.

In this section we discuss how the security policies of a service can affect its interaction with clients and how our model copes with this scenario. In particular we provide a mechanism for automatically computing the prerequisites of services and we exploit them for a simple negotiation of contracts. So far, when a service is added to an existing network we infer its type and effect and we check which other services it can safely be composed with. Now we improve our model by allowing services to specify also a prerequisite policy ψ that a session with a client has to fulfil. This policy is defined as a standard local policy, i.e. through a proper usage automaton, but its scope also includes (a part of) the client history.

Summing up, a service declares the security prerequisites for its clients as a policy ψ over the concatenation of a hypothetical client history expression and its own. Then, we partially evaluate ψ with respect to the history expression of the service obtaining the actual prerequisite $\psi^\star$. At runtime the invocation is rejected if the client history does not fulfil the requirement, i.e. it is accepted by the automaton for $\psi^\star$.

5.3.1 Security Prerequisite

Here we introduce the concept of *security prerequisite*. Basically, a prerequisite is a security requirement involving the composition between a service and its clients. In order to highlight the utility of security prerequisites we propose the following, motivating example.

Example 5.34 Many, real-world services require their clients to register to some

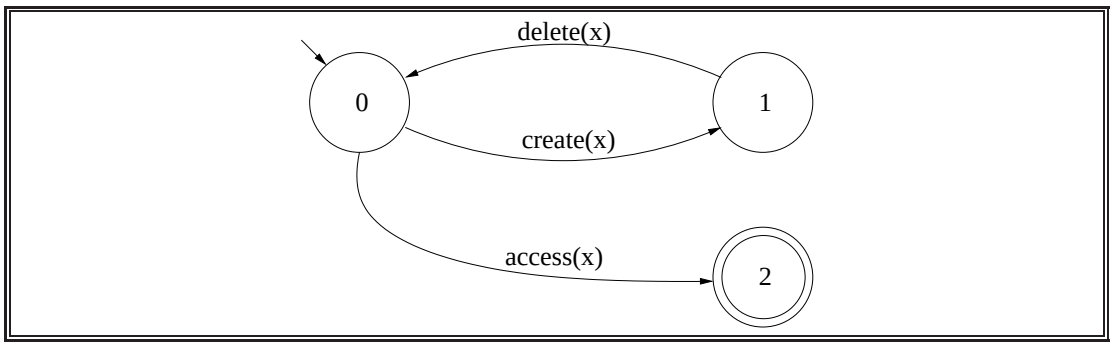


Figure 5.40: A policy rejecting unregistered clients.

identity provider beforehand. This results in a policy that also involves the behaviour of a service's clients. For instance we assume to have registered clients' identities represented by a resource of type *id*. A client can register a new *id* (action `create`) and cancel it (action `delete`) through proper services (e_c and e_d respectively). A service e, upon receiving the identity of a client, logs the access (action `access`) and serves the request (action `serve`). Suppose that the service e only accepts registered clients, so it will respect a policy ψ saying: "never permit access to unregistered clients". Actually, the service needs to inspect other parties' histories to check the policy ψ, depicted in Figure 5.40. Clearly this is infeasible under our assumptions to incrementally build services with no a priori knowledge of the potential clients.

We will see in Section 5.3.2 how to mechanically derive a prerequisite in the form of a policy that a client has to fulfil to be accepted by the server. Checking that a client satisfies $\psi^\star$ can be done at invocation time *without* running the service and *without* model checking the composition client-service, but only relying on the past history of the client.

This example will be worked out in detail in Example 5.35, where the prerequisite, called $\psi^\star$ is depicted in Figure 5.41.

Even though local policies and prerequisites are both defined using the same formalism, i.e. usage automata, we make a clear distinction between them. As a matter of fact, they differ for few, crucial aspects. As we said above, the main difference is that they apply to completely different scopes. Indeed, local policies define security properties of limited blocks of code while prerequisites extends their scope on both the client and the service.

A further difference derives from the stage of the service life-cycle they are involved in. In particular, local policies are mainly used statically for driving the creation of valid composition plans. Instead, the security prerequisites are needed dynamically for checking whether a certain history, belonging to a client, is compatible with the service invocation.

Also note that while local policies are applied through the framing operator (see

Section 5.1.2), security prerequisites are defined beside the service implementation. In this way, they are exposed at the same level of the service contract and integrate its information with the requirements that a client must fulfil.

5.3.2 Partial evaluation of policies

Partial evaluation [104] consists in specialising a program by forcing a part of its input to a constant value. Often, partial evaluation is used for the automatic generation of specialised routines and the program optimisation in compiler theory. Also, this approach proved to be suitable for the analysis of the security properties of programs. For instance, in [130] the authors combine partial evaluation and *partial model checking* [10] for mechanically building security monitors. They take advantage from the static knowledge of the system that they want to secure. In particular, they model the scenario in which a known system interacts with an unknown component. With their approach they can find a security monitor that is specialised on the system structure and is guaranteed to enforce the desired security policy on each possible component joining the system.

Here we work under similar assumptions in which a service is invoked by a client. In particular, we want that the composition of a (statically known) service with any possible client complies with a certain security requirement, i.e. the service prerequisite. The evaluation of a prerequisite involves the behaviour of the service (also including the sub-services it invokes) and the client. However, we assume that the service behaviour is statically predicted through its history expression. The goal of the partial evaluation of a service prerequisite is to find a proposition that can be checked against (the history of) a client when an invocation request arrives.

Example 5.34 illustrated how processing the history of a client can be an important aspect of the service composition. A very common and traditional mechanism for storing and handling information about the history of a client consists in using cookies. However, this is not the only approach. Briefly, services store pieces of information on their clients in order to retrieve them in the next session. Beyond local histories, services can also share their information, e.g., in the scope of a *Virtual Organisation* [90]. Our approach integrates this model by using observable actions. Indeed, actions can be both fired by a client or by other services that have been invoked before the current request.

Example 5.35 Consider the following implementation of the services outlined in Example 5.34 where the service e requires its clients to have an identity id, handled by a couple of services e_c and e_d. This prerequisite takes the form of the policy ψ in Figure 5.40.

$$e_c = \lambda x.\texttt{create}(x) \quad e_d = \lambda x.\texttt{delete}(x) \quad e = \lambda x.\texttt{access}(x);\texttt{serve}(x)$$

Clearly, ψ cannot be enforced as a local policy of one of the above services. Indeed, in that case the scope of ψ would be strictly limited to the service execution,

1: **Procedure Preq**
2: **input** H and $A_\psi = \langle \mathsf{Ev}, Q, \imath, F, T \rangle$
3: $A^R_\psi \leftarrow \langle \mathsf{Ev}, Q', \imath', F', T' \rangle$
4: $Q^\star \leftarrow \emptyset$
5: **for all** $q \in Q'$ **do**
6: $\quad A_q \leftarrow \langle \mathsf{Ev}, Q', \imath', \{q\}, T' \rangle$
7: $\quad$ **if** $\exists\, \sigma \,:\, \mathcal{L}(A_q(\sigma)) \cap [\![H^R]\!]^\sigma \neq \emptyset$ **then**
8: $\quad\quad Q^\star \leftarrow Q^\star \cup \{q\}$
9: $\quad$ **end if**
10: **end for**
11: $N_{\psi^\star} \leftarrow \langle \mathsf{Ev}, Q', Q^\star, F', T' \rangle$
12: $A_{\psi^\star} \leftarrow DFA(N^R_{\psi^\star})$
13: **return** $A_{\psi^\star}$

Table 5.16: The prerequisite synthesis algorithm

while instead it is intended to span over the history η of a client, even though the service has already been invoked and thus has no access to η.

The example above shows that making the client aware about the service requirement at an early stage avoids pointless invocation, i.e. requests that will not be served. Since we are dealing with open networks, there is no static assumption we can make on the structure and behaviour of network clients.

To cope with these issues, we extend the synthesis of composition plans with a mechanism for specifying and checking prerequisites. In our model, prerequisites work symmetrically with respect to contracts and complement the approach to service security seen so far.

In Table 5.16 we formalise the partial evaluation procedure used to deduce an actual prerequisite for the requirements put by a service on its clients. Roughly, given a service e such that $e : \tau \xrightarrow{H} \tau'$ and (the usage automaton computing) the requirement ψ, the procedure returns the partial evaluation of it with respect to H, namely the prerequisite $\psi^\star$. We start (line 3) by generating the usage automaton A^R_ψ that accepts the reverse language of A_ψ (we omit here the details and refer the interest reader to the analogous construction of [5]). Then, starting from the empty set $Q^\star$ (line 4), the procedure finds all the states of A^R_ψ that can be reached with a trace belonging to the reverse of the language of H (lines 5-10). Note that the reverse of $[\![H]\!]^\sigma$ can be computed by "reversing" the history expression H itself, similarly to what is done for reverting context-free grammars [5]. Then, we construct a sort of automaton $N_{\psi^\star}$ having $Q^\star$ as set of initial states (line 11). Clearly, as $N_{\psi^\star}$ could have more than one initial state, it is not a usage automaton. We then reverse $N_{\psi^\star}$ and convert it to a standard usage automaton $A_{\psi^\star}$ (line 12).

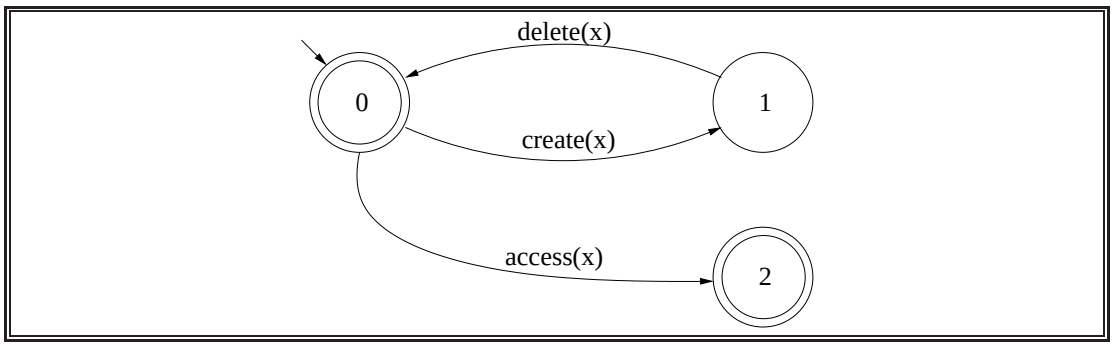


Figure 5.41: Prerequisite policy returned by $\texttt{Preq}(H, A_\psi)$

It is immediate to verify that the procedure above always terminates. Intuitively, its complexity is the same as solving the model checking problem for usage automata [29]. Indeed, in line 7, repeated for the number of states of the automaton A^R_ψ, we are intersecting a context-free language and a regular one.

The result of this algorithm is the usage automaton $A_{\psi^\star}$ that recognizes the histories violating the prerequisite $\psi^\star$. When a client invokes the service the usage policy $\psi^\star$ is checked against the actual execution history of the client. If the check is passed the client can safely invoke the service. This assertion is formalised by the following theorem.

Theorem 5.4 Let $\texttt{Preq}(H, A_\psi) = A_{\psi^\star}$ then

$$\forall\, \eta \,:\, \eta \models \psi^\star \Rightarrow \eta H \models \psi$$

□

Example 5.36 Consider again the services of Example 5.35 where the requirement of e is the policy ψ in Figure 5.40. Typing e we obtain

$$\emptyset, \varepsilon \vdash_{true} e \,:\, id \xrightarrow{H} \mathbf{1}$$

where $H = \texttt{access}(x)\texttt{serve}(x)$.

The result of the partial evaluation procedure `Preq` applied to H and A_ψ is the automaton in Figure 5.41. This automaton says that a client is accepted only if the following occurs. The client performed no `access` without being registered; it has at least one active registration, i.e. it performed at least a `create` not followed by a `delete` action. Theorem 5.4 guarantees that the client satisfying $\psi^\star$ will also respect ψ when composed with e.

Note that this mechanism is independent of the kind and source of the client data the service wants to check. For instance, the information about the clients history could be stored using cookies (on client side) or through a database shared by the services. Indeed, to apply our method we only need to know that a reliable representation of (a part of) the clients history exists and can be inspected.

5.3.3 A strategy for service orchestration

Now we show how the standard service repository model can be trivially extended in order to exploit prerequisite policies for service composition. As we mentioned in Section 5.1.2, a repository `Srv` contains the services composing the network, their types and effects as history expressions. History expressions represent the contracts that the services offer to their clients.

As we assume open networks, service developers cannot foresee the context in which their services will run. Hence, services are written according to their own specification that include their security policies and requirements on their clients, if any. When a service is actually deployed in a service network, it is processed in the following way:

1. The service code e and the prerequisite ψ (namely the automaton A_ψ) are submitted to the network orchestrator;
2. The orchestrator applies the type and effect system to e for finding its type $\tau \xrightarrow{H} \tau'$;
3. The valid composition plans are computed;
4. The prerequisite policy $\psi^\star$ is created through $\mathtt{Preq}(H, A_\psi)$.

If the above steps are successful, the pair $\langle \psi^*, e : \tau \xrightarrow{H} \tau' \rangle$ is added to `Srv` (we neglect locations here). Otherwise, the service is rejected by the orchestrator.

As said above, the actual invocation depends on the security policies of the client and the prerequisite of the service. The negotiation for a session corresponds to finding an agreement between the two. When a client requires a service, its history is checked against the prerequisite $\psi^\star$. Then the contract H is model checked against the client policies and, if successful, the invocation can safely take place.

This model can be improved in several ways. The mechanism presented above follows an "all or nothing" strategy, that is the service is accepted or rejected by the orchestrator. Nevertheless, the negotiation process can be extended by associating to each service more than one contract. As a matter of fact, we can replicate one entry of `Srv` for each valid plan available for the corresponding service. Projected over different plans, a service offers different contracts and, consequently, different prerequisite policies.

For instance, imagine to have k valid plans $\pi_1, \ldots, \pi_k$ for a service e having a prerequisite ψ. We can replace the single entry of `Srv` associated to e with the set

$$\begin{array}{c} \langle \psi_1^\star, e : \tau \xrightarrow{H_1} \tau' \rangle \\ \vdots \\ \langle \psi_k^\star, e : \tau \xrightarrow{H_k} \tau' \rangle \end{array}$$

where $\emptyset, \varepsilon \vdash_{true} e \mid_{\pi_i} : \tau \xrightarrow{H_i} \tau'$ and $A_{\psi_i^\star} = \mathtt{Preq}(H_i, A_\psi)$.

In this way clients can choose among k versions of the same service. Indeed, when a client wants to invoke a service, it asks the service repository for finding a suitable one. The repository compares the client history with the prerequisite policies of the compatible service instances, i.e. one for each viable plan. Those being satisfied are candidates for serving the request. Among them, the repository finds a contract, i.e. a history expression, that also satisfies the client policies if any. This second step consists of a model checking procedure that, in general, is much more expensive than the verification of prerequisites. In this way, we avoid solving the model checking problem for verifying the compatibility of a composition that would fail anyway.

5.4 Discussion

In this chapter we described our work on the security analysis and orchestration of service networks. The contents on this topic has been presented according to the following structure.

- Section 5.1 introduced the notion of *open network* and our computational model for web services.
- Section 5.2 presented our strategy for incrementally creating composition plans while preserving their security properties.
- Section 5.3 detailed a mechanism for the automatic synthesis of security prerequisites which lead the dynamic composition of services under security constraints.

Web services are rapidly changing the shape of the Internet. One of the reason of their success is that they offer access to arbitrary complex systems abstracting from the actual implementation. Also, they allow companies for building new services on top of existing ones, so favouring economies of scale. However, abstraction does not cope with security issues which may affect low level aspects of a service network. Hence, modern service platform should carry out the security analysis process automatically. Indeed, delegating such responsibility to service developers and designer would dramatically reduce their possibility of abstraction. The techniques presented here aim at providing service networks with security support without compromising the basic features of SOC systems.

Conclusions

We presented our contribution to providing software and service compositions with formal security guarantees. Our investigation started from a basic composition model, i.e., the integration of a mobile application in a hosting platform. In this context, we outlined the importance of including proper security mechanisms in the three main stages of the mobile applications life-cycle, i.e., *development*, *deployment* and *execution*, for providing the mobile application with formal security guarantees. We presented a framework (Sections 3.1 and 3.2) that integrates (i) the design of security policies, (ii) their application to the code, (iii) the formal verification and (iv) the automatic creation of execution monitors. Then, we addressed the complementary problem of securing a platform from an untrusted piece of mobile code. We presented an enforcement architecture for providing security assurances to a platform importing some external software (Section 3.3). Also, we showed the feasibility of our approach for the design of centralised monitors even under severe constraints on platform, e.g., on mobile devices with limited capabilities (Section 3.4).

Then, we considered a more general composition and we introduced a model that handles trust and security issues in an integrated manner. We allow for composing some agents, none of them having a complete control on the other. Under these assumptions, we added a quantitative trust management system to the Security-by-Contract standard model (Section 4.1). In our proposal, the trust evaluation is integrated among the security procedures and is responsible for deciding the security settings for a new composition. Furthermore, a new class of policy automata has been introduced (Section 4.2) for specifying and driving the new security and trust monitoring process (Section 4.3).

Finally, we moved to a very general compositional model of web services. Web services are compositional in a very general sense. Indeed, they can invoke each other through messages, remote invocations or even mobile code. In this field, we extended some existing work to deal with open networks and open sessions (Section 5.1). During our investigation, we exploited many of the techniques derived from analysis of the previous cases. Interestingly, we show that most of them are preserved through service composition under some reasonable assumptions. Moreover, we proposed an automatic approach to the secure composition of modular plans (Section 5.2) and we enhanced the call-by-contract invocation paradigm with a new element, namely the security prerequisite (Section 5.3).

Related Work

In the following we survey on some related work. For the sake of presentation, we structured the works listed below according to the three application scenarios that we detailed in this thesis. However, this organisation is quite artificial and its main purpose is to facilitate a direct comparison with our work on each specific field.

Application security. Polymer [31] is a language for specifying, composing and dynamically enforcing (global) security policies. In the lines of *edit automata* [30], a Polymer policy can intervene in the program trace to insert or suppress some events. The access control model of Java is enhanced in [146], by specifying fine-grained constraints on the execution of mobile code. A method invocation is denied when a certain condition on the dynamic state of the system is false.

Security policies are modelled as process algebras in [17, 131, 133]. There, a custom JVM is used, with an execution monitor that traps system calls and fires them concurrently to the policy. When a trapped system call is not permitted by the policy, the execution monitor tries to force a corrective event – if possible – otherwise it aborts the system call.

Since the policies of [31, 146, 17, 131, 133] are Turing-equivalent, they are very expressive, yet they have some drawbacks. First, the process of deciding if an action must be denied might not terminate. Second, non-trivial static optimizations are infeasible, unlike in our approach.

The problem of deciding whether the contract advertised by an application is compatible with that required by a mobile device is explored in [83]. To do that, a matching algorithm is proposed, based on a regular language inclusion test. In [134] the model is further extended to use *automata modulo theory*, i.e., Büchi automata where edges carry guards expressed as logical formulas. In this approach both the policy and the application behaviour are expressed using the same kind of automata. Instead, using history expressions, which have the same expressivity of context-free languages, allows for modelling richer behaviour.

The problem of wrapping method calls to make a program obey a given policy has been widely studied, and several frameworks have been proposed in the last few years. Some approaches, e.g., the Kava system [185], use bytecode rewriting to obtain behavioural run-time reflection. This amounts to modifying the structure of the bytecode, by inserting additional instructions before and after a method invocation. A different solution, adopted e.g., by JavaCloak [161], consists in exploiting the Java reflection facilities to represent Java entities through suitable behavioural abstractions. However, this is not fully applicable in our case, since currently it neither supports wrapping of constructors, nor it allows one to handle methods not defined in some interface. Note that our bytecode rewriting approach needs no such assumptions.

Many authors have studied verification techniques for history-based security at a foundational level. Static and dynamic techniques have been explored in [58, 85,

129, 180], to transform programs and make them obey a given policy. While these approaches consider global policies and no dynamic creation of objects, our model also allows for local policies, and for events parameterized over dynamically created objects.

A typed λ-calculus with primitives for creating and accessing resources, and for defining their permitted usages, is presented in [100]. A type system guarantees that well-typed programs are resource-safe. The policies of [100] can only speak about the usage of *single* resources, while ours can span over many resources.

A lot of effort has been devoted to develop verification techniques for Java programs. The Soot project [175] provides a comprehensive Java optimization framework, also exploiting static analysis techniques to compute approximated data flow and control flow graphs, and points-to information. In [38] Java source programs are monitored through *tracematches*, a kind of policies that constrain the execution traces of method calls. Static analysis is used to prove that code adheres to the tracematch at hand, so the monitor can be removed. Unlike ours, these policies are global. Also, bytecode analysis is not considered.

JACK [20] is a tool for the validation of Java applications, both at the levels of bytecode and of source code. Programmers specify application properties through JML annotations, which are as expressive as first-order logic. These annotations give rise to proof obligations, to be statically verified by a theorem prover. The verification process might require the intervention of the developer to resolve the proof obligations, while in our framework the verification is fully automated.

A security study of Java ME has been presented by Kolsi and Virtanen in [108], where they described the possible threats and the security needs in a mobile environment. In particular, they described how MIDP 2.0 solved some security issues of MIDP 1.1, but they concluded that some issues are still present. A security analysis of Java ME has been presented also by Debbabi et al. in [72, 73, 74]. In these papers, they detail the MIDP and CLDC security architecture, and they identify a set of vulnerabilities of this architecture. Moreover, they also test some attack scenarios on actual mobile phones. However, the previous papers do not propose any improvement to the Java ME security support to solve the security issues they described.

In [79] the authors present a framework for the run time monitoring of applications running on the .NET platform exploiting the in-lining technique. Their model is symmetrical to the one we have presented in Section 3.3 but for the technical differences deriving from the diverse environment. Recently, they also presented in [78] a complete system composed by an instrumentator for modifying the .NET Intermediate Language and a PDP for enforcing ConSpec policies. Actually, the proposed framework implements the PDP as a *Dynamic Linked Library* that is invoked during the execution. Instead, our model exploits a separate process for monitoring the running applications. The necessity for this mechanism arises from the different execution environment. Indeed the Java MIDlets do not share the execution context and they can not access public, external libraries.

In [138] the author presents *proof carrying code* (PCC). PCC generalises the idea of code signature to a formal proof that is attached to certified applications. Roughly, the producer of the MIDlet provides an automatically computed proof that the final user can verify against the received code. While the proof extraction requires some computational effort, verifying the compliance between the certificate and the MIDlet is a quite simple task. Hence this approach is particularly suitable for the context of mobile devices and frameworks based on it have been presented, e.g., [35]. However, there are two main issues that we must consider comparing PCC with our system: proofs size and policies customization. In general, expressive security policies can lead to long certificates. Since each MIDlet must be signed, we pay this memory overhead several times. Unlike, our system works with only one copy of the enforced policy. Moreover, when using PCC it is necessary that users and producers agree on the security properties used for signing the MIDlets. This in a severe constraint for both the vendors and the costumers. Indeed, the first must provide very general and possibly complex certificates. The second, can not change the security policies unless they replace the installed MIDlets with a new, policy-compliant copy of them.

Security and trust. Recently, the full integration between security enforcement mechanisms and trust management systems is receiving more attention. For instance, some work has been done for adding a trust management system to fine-grained access control in Grid Architecture. A proposal can be found in [111] where the authors present an access control system that enhances the Globus toolkit with a number of features. Among them, trust-based decisions are used to drive the interactions among the grid nodes.

Along this line of research [60] presents an integrated architecture, extending the previous one, with an inference engine managing reputation and trust credentials. This framework is extended again in [110] where a mechanism for trust negotiating credential is introduced to overcome some scalability issues. In this way the framework preserves privacy credentials and the security policies of both users and providers. Even if the application scenario and the implementation are different, the basic idea consists in considering trust metrics for deciding the reliability of an application provider.

Also [125] presents a reputation mechanism to facilitate the trustworthiness evaluation in ubiquitous computing environments. This method is based on probability theory and supports reputation evolution and propagation. The proposed reputation mechanism is also implemented as part of a QoS-aware Web service discovery middleware and evaluated against its overhead on service discovery latency. It is important to note that our approach is not probabilistic. Indeed, according to our method the responsibility of the trustworthiness decision is deterministically affected by the applications behaviour.

In [97] the authors show a method for combining trust management theories

with nonce-based cryptographic protocols. Roughly, they use formulas from a trust management logic for annotating the transmit and receive actions of the protocol principals following an assume-guarantee reasoning. Basically, their approach deals with the correctness of cryptographic protocols using a well defined model of trust. Instead, our proposal does not depend on any predefined trust model.

Security in service composition. As said, our contribution on secure service composition builds upon the work of Bartoletti et al. [23, 26, 28]. We have compared our results with theirs along Chapter 5, in particular in Section 5.1.

In [42] the authors present a framework for contract-based creation of choreographies. Roughly, they use a contract system for finding a match between contracts and choreographies. In this way they verify whether a given contract, declared by a service joining the network, is consistent with the current choreography. However, this framework exploits a global knowledge about the network structure while our model also deals with open networks.

Among the formal approaches to the description of web services and their interactions presented in the last years, CaSPiS [44] is a major proposal. Basically, sessions are permitted only when a client and a service agree on a security level. This check is done through a security-oriented type system [109]. Well typed processes are free from a finite set of common security errors that could arise at runtime. The main difference with respect to our work is that here the security properties are fixed, while in our approach they are user defined.

A process algebra for service orchestration is proposed in [18]. Briefly, the authors propose an extension of the interfaces of services with a behavioural pattern. Patterns are then used for computing a description of the system arising from a service composition. Even though patterns can be seen as security requirements, they must be known when the composition is computed, which is equivalent to having a closed network.

Castagna et al. [51] use a variant of CCS for defining service contracts. A subcontract relation guarantees that the choreography always respects the contracts of both client and service. This approach assumes that clients always know the request contract and it is focused on finding a satisfactory composition with some available service. Since the composition depends on the possible instantiations of a service, such a contract could be unavailable during the analysis phase. Instead we produce valid plans independently of the actual instantiation of resources.

Busi et al. [49] propose an analysis of service orchestration and choreography. In their work, the validity of the service orchestration is a consequence of its conformance with respect to the intended choreography. The main difference with our work consists in the approach to choreography. Indeed, they start from a pre-defined choreography and verify the validity of an orchestration. Instead, our approach aims at checking partial service composition without relying on any global orchestrator.

In [130] a framework for the synthesis of orchestrators is presented. This tech-

nique consists in automatically producing an orchestrator guaranteeing that the service composition respects the desired security policy. This approach defines a composition that complies with the policy of a client. Since our method produces partial compositions that respect all the involved policies (of both clients and servers), it seems to be more general. However, as [130] generates dynamic orchestrators, that system works under completely different assumptions with respect to ours.

Future Work

We see many possible directions for extending the present work. Actually, some of them are already under investigation and some preliminary results have been achieved. For instance, in [67] we presented a new logic, namely *elective temporal logic* (ETL), for the specification of the temporal properties of programs. The ETL specifications can be used for the verification of programs and, even more important, they can drive optimised execution monitors. Indeed, a controller using a ETL specification can safely predict the minimum number of actions that, starting from the current state, can lead to a violation. In this way, the monitor can suspend its activity for the predicted amount of time with no risk of a policy violation.

Beyond the obvious benefit of reducing the monitoring activity, we aim at extending this line of research for finding a new formalisation of *probabilistic security*. As a matter of fact, in many practical cases, we cannot assume a security monitor to simply follow the execution of its target action by action. Sometimes it is costly observing an action and we would prefer to reduce such overhead. Also, it is often the case that the monitor cannot be proactive, i.e. it cannot suspend an action for running security checks. Probabilistic monitors can cope with these issues. Indeed, they can skip unnecessary checks and focus only on really critical operations. For doing that, such kind of monitor also considers a risk factor, i.e. the probability of failing to prevent a security violation. Through the combination of probabilistic monitors and probabilistic model checkers, with aim at defining a more hopefully complete picture of probabilistic security.

Following the approach of Section 4.1, we are also interested in studying the possibility of a trust-based composition of services. As a matter of fact, many usages of web services are affected or even based on concepts like trust and reputation. Even though several advancements have been done on how to provide service compositions with security guarantees, often the providers and their customers still prefer to rely on their informal opinions about the counterparts. Perhaps, this behaviour depends on the partial misunderstanding of the security mechanisms. Indeed, the formal specification of security requirements is still too technical. Hence, we think that these methods should be integrated with high level facilities for dealing with the human perception of the security issues.

Bibliography

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46:749–786, September 1999.

[2] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings 10th Annual Network and Distributed System Security Symposium*, 2003.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *4th ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[4] S. Adams. Children get first mobile phone at average age of eight. `http://www.telegraph.co.uk/`, February 2009.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of the First Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 07)*, Oslo, Norway, October 2007.

[7] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 07)*, Dresden, Germany, September 2007. ESORICS.

[8] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2:117–126, 1986.

[9] G. Amato, P. Bolettieri, G. Costa, F. L. Torre, and F. Martinelli. Detection of images with adult content for parental control on mobile devices. In *Proceedings of the 6th International Conference on Mobile Technology, Applications and Systems, Mobility Conference*. ACM, 2009.

[10] H. R. Andersen. Partial Model Checking (Extended Abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[11] S. Anderson et al. *Web Services Trust Language (WS-Trust)*, 2005. http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf.

[12] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1, 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[13] Android Market, 2011. https://market.android.com/.

[14] App Store for iPhone, 2011. http://www.apple.com/.

[15] M. Archer and C. L. Heitmeyer. Human-Style Theorem Proving Using PVS. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '97, pages 33–48, London, UK, 1997. Springer-Verlag.

[16] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Prentice Hall, forth edition, August 2005.

[17] F. Baiardi, F. Martinelli, P. Mori, and A. Vaccarelli. Improving grid services security with fine grain policies. In *OTM Workshops*, 2004.

[18] M. A. Barbosa and L. S. Barbosa. A perspective on service orchestration. *Science of Computer Programming*, 74(9):671–687, 2009.

[19] L. Baresi, S. Guinea, and P. Plebani. WS-Policy for Service Monitoring. In C. Bussler and M.-C. Shan, editors, *Technologies for E-Services*, volume 3811 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 2006.

[20] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. Jack - a tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects, 5th International Symposium, Amsterdam, NL, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2007.

[21] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino. Securing Java with Local Policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[22] M. Bartoletti, G. Costa, and R. Zunino. Jalapa: Securing Java with Local Policies. *Electronic Notes in Theoretical Computer Science*, 253(5):145–151, 2009.

[23] M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing Secure Service Composition. In *Proceedings of the 18th Computer Security Foundations Workshop (CSFW)*, 2005.

[24] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based Access Control with Local Policies. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, 2005.

[25] M. Bartoletti, P. Degano, and G. L. Ferrari. Policy Framings for Access Control. In *Workshop on Issues in the Theory of Security*, 2005.

[26] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Secure service orchestration. In *Foundations of Security Analysis and Design*, pages 24–74, 2007.

[27] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. Technical report, Università di Pisa, 2008.

[28] M. Bartoletti, P. Degano, G.-L. Ferrari, and R. Zunino. Local policies for resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 31(6):1–43, 2009.

[29] M. Bartoletti and R. Zunino. Locust: a tool for model checking usage policies. Technical report, Università di Pisa, 2008.

[30] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS)*, 2002.

[31] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 305–314. ACM, 2005.

[32] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE Corporation, 1973.

[33] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[34] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 76–87, New York, NY, USA, 2002. ACM.

[35] F. Besson, G. Dufay, and T. Jensen. A Formal Model of Access Control for Mobile Interactive Devices. 2006.

[36] F. Besson, T. P. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[37] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[38] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[39] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer Berlin / Heidelberg, 2008.

[40] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. *Web services security (WS-Security)*, 2002.

[41] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1 Specification, 2000. `http://www.w3.org/TR/soap/`.

[42] M. Bravetti, I. Lanese, and G. Zavattaro. Contract-driven implementation of choreographies. In *Proceedings of the 4th International Symposium on Trustworthy Global Computing*, pages 1–18, 2008.

[43] M. Bravetti and G. Zavattaro. A foundational theory of contracts for multiparty service composition. *Fundamenta Informaticae*, 89:451–478, December 2008.

[44] R. Bruni. Calculi for service-oriented computing. In *9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, pages 1–41, 2009.

[45] R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in soc. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin / Heidelberg, 2008.

[46] J. Brzosowski and J. E. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. on Electronic Computers*, 1963.

[47] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[48] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *International Journal on Sofrware Tools for Technology Transfer*, 7(3):212–232, 2005.

[49] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *Proceedings of 3rd International Conference on Service Oriented Computing*, pages 228–240, 2005.

[50] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proceedings of the 16th European conference on Programming*, pages 2–17, Berlin, Heidelberg, 2007. Springer-Verlag.

[51] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *Journal ACM Transactions on Programming Languages and Systems*, 31:1–61, July 2009.

[52] A. Castrucci, F. Martinelli, P. Mori, and F. Roperti. Enhancing Java ME Security Support with Resource Usage Monitoring. In *ICICS*, pages 256–266, 2008.

[53] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Simple Object Access Protocol (SOAP) 1.1 Specification, 2001. `http://www.w3.org/TR/wsdl/`.

[54] Google Chrome Web Store, 2011. `https://chrome.google.com/webstore`.

[55] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[56] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[57] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[58] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2000.

[59] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[60] M. Colombo, F. Martinelli, P. Mori, M. Petrocchi, and A. Vaccarelli. Fine grained access control with trust and reputation management for globus. In *OTM Conferences (2)*, pages 1505–1515, 2007.

[61] G. Costa, P. Degano, and F. Martinelli. Secure Service Composition with Symbolic Effects. In *Proceedings of the 4th South-East European Workshop on Formal Methods*, pages 3–9, 2009.

[62] G. Costa, P. Degano, and F. Martinelli. Modular plans for secure service composition. *Journal of Computer Security*, 2011. To Appear.

[63] G. Costa, P. Degano, and F. Martinelli. Secure service orchestration in open networks. *Journal of Systems Architecture - Embedded Systems Design*, 57(3):231–239, 2011.

[64] G. Costa, N. Dragoni, A. Lazouski, F. Martinelli, F. Massacci, and I. Matteucci. Extending Security-by-Contract with Quantitative Trust on Mobile Devices. In *CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 872–877, 2010.

[65] G. Costa, A. Lazouski, F. Martinelli, I. Matteucci, V. Issarny, R. Saadi, N. Dragoni, and F. Massacci. Security-by-Contract-with-Trust for Mobile Devices. *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications (JoWUA)*, 1:75–91, December 2010.

[66] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter. Runtime monitoring for next generation Java ME platform. *Computers & Security*, 29(1):74–87, 2010.

[67] G. Costa and I. Matteucci. Elective Temporal Logic. In *Proceedings of 2nd International ACM Sigsoft Symposium on Architecting Critical Systems*, June 2011. To Appear.

[68] G. Costa and I. Matteucci. Trust-Driven Policy Enforcement through Gate Automata. In *Proceedings of the 5th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011. To Appear.

[69] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2010.

[70] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security Monitor Inlining for Multithreaded Java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 546–569, Berlin, Heidelberg, 2009. Springer-Verlag.

[71] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[72] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Java for mobile devices: A security study. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC05)*, pages 235–244. IEEE Computer Society, 2005.

[73] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Security analysis of mobile Java. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications, 2005*, pages 231– 235. IEEE Computer Society, 2005.

[74] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Security evaluation of J2ME CLDC embedded Java platform. *Journal of Object Technology*, 2(5):125–154, 2006.

[75] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. *Web services security policy language (WS-SecurityPolicy)*, 2002.

[76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, May 1976.

[77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[78] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS .NET run time monitoring, March 2009. BYTECODE '09.

[79] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-Contract on the .NET platform. *Information Secururity Tech. Report*, 13(1):25–32, 2008.

[80] K. Donnelly, J. J. Hallett, and A. Kfoury. Formal semantics of weak references. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management*, 2006.

[81] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard. Security-by-Contract (S×C) for Software and Services of Mobile Systems. In *At your service: Service Engineering in the Information Society Technologies Program*. MIT press, 2009.

[82] N. Dragoni and F. Massacci. Security-by-Contract for web services. In *Proceedings of the 2007 ACM workshop on Secure web services*, pages 90–98, New York, NY, USA, 2007. ACM.

[83] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *EuroPKI*, volume 4582 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2007.

[84] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[85] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 7th New Security Paradigms Workshop*, 1999.

[86] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, page 246, Oakland, California, USA, May 2000. IEEE Computer Society.

[87] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.

[88] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Notices*, 37:234–245, May 2002.

[89] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM.

[90] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15:200–222, August 2001.

[91] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[92] G. Gheorghe, P. Mori, B. Crispo, and F. Martinelli. Enforcing ucon policies on the enterprise service bus. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems, OTM 2010*, volume 6427 of *Lecture Notes in Computer Science*, pages 876–893. Springer Berlin / Heidelberg, 2010.

[93] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.

[94] Google TV, 2011. `http://www.google.com/tv/`.

[95] P. Greci, F. Martinelli, and I. Matteucci. A framework for contract-policy matching based on symbolic simulations for securing mobile device application. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 221–236. Springer Berlin Heidelberg, 2009.

[96] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.

[97] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust Management in Strand Spaces: A Rely-Guarantee Method. In *Proceedings of the 13th European Symposium on Programming*, Lecture Notes in Computer Science, pages 325–339. Springer, 2004.

[98] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[99] iCareMobile official website, 2011. `http://icaremobile.iit.cnr.it/`.

[100] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[101] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[102] Jalapa: Securing Java with Local Policies web page at Sourceforge.net, 2008. `http://jalapa.sourceforge.net/`.

[103] Java + Information Flow web page, 2011. `http://www.cs.cornell.edu/jif/`.

[104] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[105] JSR 118 Expert Group. Mobile Information Device Profile for Java 2 Micro Edition. Java Standards Process JSP 118, Java Community Process, `http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html`, November 2002.

[106] K. Keahey and V. Welch. Fine-grain authorization for resource management in the grid environment. In *Proceedings of the Third International Workshop on Grid Computing*, pages 199–206, London, UK, 2002. Springer-Verlag.

[107] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of WWW (Alternate Paper Tracks)*, 2003.

[108] O. Kolsi and T. Virtanen. MIDP 2.0 security enhancements. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, 2004.

[109] M. Kolundzija. Security types for sessions and pipelines. In *5th International Workshop on Web Services and Formal Methods*, pages 175–190, 2008.

[110] H. Koshutanski, A. Lazouski, F. Martinelli, and P. Mori. Enhancing grid security by fine-grained behavioral control and negotiation-based authorization. *International Journal of Information Sececurity*, 8(4):291–314, 2009.

[111] H. Koshutanski, F. Martinelli, P. Mori, L. Borz, and A. Vaccarelli. A Fine Grained and X.509 Based Access Control System for Globus. In *OTM*, pages 1336–1350. Springer, 2006.

[112] H. Koshutanski, F. Martinelli, P. Mori, and A. Vaccarelli. Fine-grained and History-based Access Control with Trust Management for Autonomic Grid Services. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 34, Washington, DC, USA, 2006. IEEE Computer Society.

[113] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[114] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[115] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, November 1992.

[116] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8:18–24, January 1974.

[117] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *IEEE International Conference on Software Engineering and Formal Methods*, pages 305–314, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[118] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[119] K. R. M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[120] A. Lenhart. Teens and mobile phones over the past five years: Pew internet looks back. `http://www.pewinternet.org/`, August 2009.

[121] A. Lenhart. Teens and sexting. `http://www.pewinternet.org/`, December 2009.

[122] A. Lenhart, R. Ling, S. Campbell, and K. Purcell. Teens and mobile phones. `http://www.pewinternet.org/`, April 2010.

[123] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[124] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[125] J. Liu and V. Issarny. An incentive compatible reputation mechanism for ubiquitous computing environments. *International Journal of Information Security*, 6(5):297–311, 2007.

[126] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, January 1998.

[127] G. Lowe. Analysing Protocol Subject to Guessing Attacks. *Journal of Computer Security*, 12(1):83–98, 2004.

[128] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[129] K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Programming Languages Symposium*, 2003.

[130] F. Martinelli and I. Matteucci. Through Modeling to Synthesis of Security Automata. *Electronic Notes in Theoretical Compututer Science*, 179:31–46, 2007.

[131] F. Martinelli and P. Mori. Enhancing java security with history based access control. In *FOSAD*, pages 135–159, 2007.

[132] F. Martinelli, P. Mori, T. Quillinan, and C. Schaefer. A runtime monitoring environment for mobile Java. In *Proceedings of the 1st International ICST workshop on Security Testing (SecTest08)*, 2008.

[133] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *ICAS/ICNS*, 2005.

[134] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.

[135] I. Mastroeni. On the Role of Abstract Non-interference in Language-Based Security. In K. Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433. Springer Berlin / Heidelberg, 2005.

[136] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[137] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[138] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[139] F. Nielson, R. R. Hansen, and H. R. Nielson. Abstract interpretation of mobile ambients. *Science of Computer Programming*, 47:145–175, May 2003.

[140] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.

[141] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[142] C. Nodder. *Cranor, L. and Garfinkel, S. L., Designing Secure Systems That People Can Use*, chapter Users and Trust: A Microsoft Case Study. O'Reilly & Associates, 2005.

[143] OpenMoko project. OpenMoko. `http://openmoko.org`, 2010.

[144] OSGi Alliance Web Page. Simple Object Access Protocol (SOAP) 1.1 Specification, 2001. `http://www.w3.org/TR/wsdl/`.

[145] Nokia OVI Store, 2011. `http://store.ovi.com/`.

[146] R. Pandey and B. Hashii. Providing fine-grained access control for java programs. In *ECCOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 449–473. Springer, 1999.

[147] M. P. Papazoglou. *Web Services: Principles and Technology.* Prentice Hall, 1999.

[148] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, 2003.

[149] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing: A research roadmap. In F. Cubera, B. J. Krämer, and M. P. Papazoglou, editors, *Service Oriented Computing (SOC)*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[150] M. Papazouglou and D. Georgakopoulos. Special issue on service oriented computing. *Communications of the ACM*, 46(10), 2003.

[151] J. Park and R. Sandhu. The UCON_{ABC} usage control model. *ACM Transactions on Information and System Security*, 7:128–174, February 2004.

[152] A. S. Patrick, P. Briggs, and S. Marsh. *Designing Secure Systems That People Can Use*, chapter Designing Systems that People will Trust. O'Reilly & Associates, 2005.

[153] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[154] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[155] Pew Research Center. Pew Internet and American Life Project. `http://www.pewinternet.org/`, August 2011.

[156] Pew Research Center. Pew Research Center web site. `http://www.pewinternet.org/`, 2011.

[157] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, November 1977.

[158] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Commununications of the ACM*, 49:39–44, September 2006.

[159] Proteus J2ME Browser. `http://sourceforge.net/projects/protheus/`, 2010.

[160] Python programming language official website, 2011. `http://www.python.org/`.

[161] K. V. Renaud. Experience with statically-generated proxies for facilitating Java runtime specialisation. *IEEE Proc. Software*, 149(6), Dec 2002.

[162] A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 772–786. Springer Berlin / Heidelberg, 1997.

[163] Ruby programming language official website, 2011. `http://www.ruby-lang.org/`.

[164] S3MS project. Security for Software and Services for Mobile Systems (S3MS). `http://www.s3ms.org`.

[165] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati. Automated SLA Monitoring for Web Services. In M. Feridun, P. Kropf, and G. Babin, editors, *Management Technologies for E-Commerce and E-Business Applications*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer Berlin / Heidelberg, 2002.

[166] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.

[167] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32, 1994.

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

[169] P. Schnoebelen. The complexity of temporal logic model checking. In *Proceedings of the 4th Workshop on Advances in Modal Logic (AIML'02)*, 2003.

[170] R. Sekar, V. 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*, SOSP '03, pages 15–28, New York, NY, USA, 2003. ACM.

[171] P. Sewell and J. Vitek. Secure composition of untrusted code: box-π, wrappers, and causality types. *Journal of Computer Security*, 11:135–187, March 2003.

[172] C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.

[173] C. Skalka, S. F. Smith, and D. V. Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[174] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.

[175] Soot: a Java optimization framework `http://www.sable.mcgill.ca/soot/`.

[176] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 3rd edition, 2002.

[177] Sun Microsystems. White Paper on KVM and the Connected, Limited Device Configuration (CLDC). Technical report, Palo Alto, CA, USA, May 2000.

[178] Sun Microsystems Inc. The Connected Limited Device Configuration Specification. Java Standards Process JSR 139, `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`, March 2003.

[179] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3rd edition, 2007.

[180] P. Thiemann. Enforcing safety properties using type specialization. In *Proc. ESOP*, 2001.

[181] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, January 1996.

[182] D. Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2000.

[183] I. Walukiewicz. *A complete deductive system for the μ-calculus*. PhD thesis, Warsaw University, 1994.

[184] I. Welch and R. Stroud. Kava - A reflective Java based on Bytecode rewriting. *Reflection and Software Engineering*, pages 155–167, 2000.

[185] I. Welch and R. Stroud. Kava - Using Bytecode rewriting to add behavioural reflection to Java. In *Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, 2001.

[186] T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[187] World Wide Web Consortium (W3C). The "Buy Something" scenario. `http://www.w3.org/2001/03/WSWS-popa/paper51`, 2009.

[188] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Transactions on Programming Languages and Systems*, 8:351–387, November 2005.

[189] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, SACMAT '04, pages 1–10, New York, NY, USA, 2004. ACM.

Appendix

Technical Proofs of Chapter 5

Lemma 5.1 If $\Gamma, H \vdash_g e : \tau$ and $g' \Rightarrow g$ then $\Gamma, H \vdash_{g'} e : \tau$ □

Proof. By induction on the structure of e.

- Cases for $e = *$, $e = r$ and $e = x$, trivial.
- If $e = \alpha(e')$ then, applying the inductive hypothesis to the premise of the rule we have

$$\frac{\Gamma, H \vdash_{g'} e' : \mathcal{R}}{\Gamma, H \cdot \sum_{\mathcal{R}} \alpha(r) \vdash_{g'} \alpha(e') : \mathbf{1}}$$

- If $e = \texttt{if}\ \bar{g}\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff}$ we note that for all $g, g', \bar{g}$ $g' \Rightarrow g$ implies that $g' \wedge \bar{g} \Rightarrow g \wedge \bar{g}$. The thesis follows by applying the inductive hypothesis to the premises of the rule.
- If $e = \varphi[e']$ the property follows directly from the inductive hypothesis.
- If $e = \lambda_z x.e'$ we apply the inductive hypothesis and we immediately have

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_{g'} e' : \tau'}{\Gamma, \varepsilon \vdash_{g'} \lambda_z x.e' : \tau \xrightarrow{H} \tau'}$$

- If $e = e_1 e_2$ we simply apply the inductive hypothesis to e_1 and e_2. We finish using the typing rule for application.
- If $e = \texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$ the thesis follows directly from the definition of the rule $(\texttt{T} - \texttt{Req})$.

□

We introduce here a new type system for λ^{req}. We need a new type system because the rules defined in Section 5.1.3 (see Table 5.11) do not apply to terms containing some open security framing, that is a framing that has been activated (identified by the marker m).

The new typing relation $\vdash^\sharp$ has the same rules as $\vdash$ (we use T2− instead of T− to identify them). Moreover, $\vdash^\sharp$ has a new rule for framing:

$$(\texttt{T2}-\texttt{Frm}_1)\ \frac{\Gamma, H \vdash^\sharp_g e : \tau}{\Gamma, H\cdot]^m_\varphi \vdash^\sharp_g \varphi^m[e] : \tau}$$

Property 5.1 If $\Gamma, H \vdash_g e : \tau$ then $\Gamma, H \vdash^\sharp_g e : \tau$.

Proof. By definition of $\vdash^\sharp$. □

Property 5.2 If $\Gamma, H \vdash_g e : \tau$ then for each $\eta \in [\![H]\!]^\sigma$ it holds that $\eta^\partial = \eta$.

Proof. By definition of $\vdash$ and ∂. □

Property 5.3 Let $\Gamma, H \vdash^\sharp_g e : \tau$ and $\eta, e \to_\pi \eta', e'$. For each g' such that $g' \Rightarrow g$, there exists H' such that $\Gamma, H' \vdash^\sharp_{g'} e' : \tau$ and $\forall\sigma.\sigma \models g' \Longrightarrow (\eta'[\![H']\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$

Proof. By induction on the depth of $\Gamma, H \vdash^\sharp_g e : \tau$. Note that cases (T2−Unit), (T2−Res) and (T2−Var) cannot be used as base for the induction since they do not admit transitions.

- Case (T2−Ev). We have two sub-cases

  a) $\eta, \alpha(r) \to_\pi \eta\alpha(r), *$. We instantiate the rule $(\texttt{T2}-\texttt{Ev})$ to

  $$\frac{\Gamma, \varepsilon \vdash^\sharp_g r : \mathcal{R}}{\Gamma, \sum_{\mathcal{R}} \alpha(r) \vdash^\sharp_g \alpha(r) : \mathbf{1}}$$

  We simply note that $\forall\sigma.\eta\alpha(r)[\![\varepsilon]\!]^\sigma \subseteq \eta[\![\sum_{\mathcal{R}} \alpha(r)]\!]^\sigma$.

  b) $\eta, \alpha(\bar{e}) \to_\pi \eta', \alpha(\bar{e}')$. We instantiate the hypothesis to

  $$\frac{\Gamma, H \vdash^\sharp_g \bar{e} : \mathcal{R}}{\Gamma, H \cdot \sum_{\mathcal{R}} \alpha(r) \vdash^\sharp_g \alpha(\bar{e}) : \mathbf{1}}$$

  and

  $$\frac{\eta, \bar{e} \to_\pi \eta', \bar{e}'}{\eta, \alpha(\bar{e}) \to_\pi \eta', \alpha(\bar{e}')}$$

  Assuming the premises of the two rules and applying the inductive hypothesis we infer that for each $\bar{g}'$ s.t. $\bar{g}' \Rightarrow g$ then $\Gamma, \bar{H}' \vdash^\sharp_{\bar{g}'} \bar{e}' : \mathcal{R}$ implies that $\forall\sigma.\sigma \models \bar{g}' \Longrightarrow (\eta'[\![\bar{H}']\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$. Applying the typing rule for events we have

  $$\frac{\Gamma, \bar{H}' \vdash^\sharp_{\bar{g}'} \bar{e}' : \mathcal{R}}{\Gamma, \bar{H}' \cdot \sum_{\mathcal{R}} \alpha(r) \vdash^\sharp_{\bar{g}'} \bar{e}' : \mathcal{R}}$$

  Then $\forall\sigma.\sigma \models \bar{g}' \Longrightarrow (\eta'[\![\bar{H}']\!]^\sigma[\![\sum_{\mathcal{R}} \alpha(r)]\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma[\![\sum_{\mathcal{R}} \alpha(r)]\!]^\sigma)^\partial$.

- Case (T2−If). We have two symmetric cases. Instantiating the rule we obtain

$$\frac{\Gamma, H \vdash^\sharp_{g \wedge \bar{g}} e_{tt} : \tau \quad \Gamma, H \vdash^\sharp_{g \wedge \neg \bar{g}} e_{ff} : \tau}{\Gamma, H \vdash^\sharp_g \texttt{if}\ \bar{g}\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} : \tau}$$

and

$$\eta, \texttt{if}\ \bar{g}\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \rightarrow_\pi \eta, e_{\mathcal{B}(\bar{g})}$$

By inductive hypothesis we have that $\forall g'$ such that $g' \Rightarrow g \wedge \bar{g}$ the property holds. We then conclude by observing that if $g' \Rightarrow g \wedge \bar{g}$ then $g' \Rightarrow g$.

- Case (T2−Frm). Instantiating the premises we have

$$\frac{\Gamma, \bar{H} \vdash^\sharp_g \bar{e} : \tau}{\Gamma, [_\varphi \bar{H}]_\varphi \vdash^\sharp_g \varphi[\bar{e}] : \tau}$$

and

$$\eta, \varphi[\bar{e}] \rightarrow_\pi \eta[^m_\varphi, \varphi^m[\bar{e}]$$

The, typing $\varphi^m[\bar{e}]$ we have

$$\frac{\Gamma, \bar{H} \vdash^\sharp_g \bar{e} : \tau}{\Gamma, \bar{H}]^m_\varphi \vdash^\sharp_g \varphi^m[\bar{e}] : \tau}$$

We conclude by observing that $(\eta[^m_\varphi \cdot [\![\bar{H}]\!]^\sigma \cdot]^m_\varphi)^\partial = (\eta[_\varphi \cdot [\![\bar{H}]\!]^\sigma \cdot]_\varphi)^\partial$.

- Case (T2−App). Let $e = e_1 e_2$. We have

$$\frac{\Gamma, H_0 \vdash^\sharp_g e_1 : \tau \xrightarrow{H_2} \tau' \quad \Gamma, H_1 \vdash^\sharp_g e_2 : \tau}{\Gamma, H_0 \cdot H_1 \cdot H_2 \vdash^\sharp_g e_1 e_2 : \tau'}$$

We must verify four possible sub-cases depending on the rule used to derive $\eta, e \rightarrow_\pi \eta', e'$.

  - If (S−App$_1$) has been used, then

$$\frac{\eta, e_1 \rightarrow_\pi \eta', e_1'}{\eta, e_1 e_2 \rightarrow_\pi \eta', e_1' e_2}$$

    Applying the inductive hypothesis to e_1 we infer that $\forall \bar{g}$ such that $\bar{g} \Rightarrow g$ the property holds on $\Gamma, \bar{H} \vdash^\sharp_{\bar{g}} e_1' : \tau \xrightarrow{H_2} \tau'$. Then, by lemma 5.1 and property 5.1, we know that $\Gamma, H_1 \vdash^\sharp_{\bar{g}} e_2 : \tau$. So, we apply (T2-App) and we have

$$\frac{\Gamma, \bar{H} \vdash^\sharp_{\bar{g}} e_1' : \tau \xrightarrow{H_2} \tau' \quad \Gamma, H_1 \vdash^\sharp_{\bar{g}} e_2 : \tau}{\Gamma, \bar{H} \cdot H_1 \cdot H_2 \vdash^\sharp_{\bar{g}} e_1' e_2 : \tau'}$$

    Since $(\eta' [\![\bar{H}]\!]^\sigma)^\partial [\![H_1 \cdot H_2]\!]^\sigma \subseteq (\eta [\![H]\!]^\sigma)^\partial [\![H_1 \cdot H_2]\!]^\sigma$, the thesis follows.

– If ($\mathtt{S{-}App_2}$) has been used, then $e = ve_2$ and

$$\frac{\eta, e_2 \to_\pi \eta', e_2'}{\eta, ve_2 \to_\pi \eta', ve_2'}$$

We proceed similarly to the previous case but we apply the inductive hypothesis to e_2 and lemma 5.1 and property 5.1 to v. Then we obtain

$$\frac{\Gamma, \varepsilon \vdash^\sharp_{\bar{g}} v : \tau \xrightarrow{H_2} \tau' \quad \Gamma, \bar{H} \vdash^\sharp_{\bar{g}} e_2' : \tau}{\Gamma, \varepsilon \cdot \bar{H} \cdot H_2 \vdash^\sharp_{\bar{g}} ve_2' : \tau'}$$

We conclude noting that $\forall\sigma.(\eta'[\![\varepsilon \cdot \bar{H}]\!]^\sigma)^\partial = (\eta'[\![\bar{H}]\!]^\sigma)^\partial$.

– ($\mathtt{S{-}App_3}$) has been used, then $e = (\lambda_z x.e')v$ and

$$\eta, (\lambda_z x.e')v \to_\pi \eta, e'\{v/x, \lambda_z x.e'/z\}$$

We finish by applying the same reasoning as in the previous cases.

– If ($\mathtt{S{-}Req}$) has been used, we have

$$\frac{e_{\bar{\ell}} : \tau \xrightarrow{H^*} \tau' \in \mathtt{Srv}\rfloor_\ell \qquad \pi(\rho) = \bar{\ell} \qquad H^* \models \varphi}{\eta, (\mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau')v \to_\pi \eta, e_{\bar{\ell}} v}$$

By ($\mathtt{T2{-}Req}$) follows that

$$\frac{I = \{H \mid e_{\ell'} : \tau \xrightarrow{H} \tau' \in \mathtt{Srv}\rfloor_\ell \wedge H \models \varphi\}}{\Gamma, \varepsilon \vdash_g \mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau' : \tau \xrightarrow{\sum_{i\in I} H_i} \tau'}$$

We conclude by noting $\forall\sigma.\forall\, \hat{H}.(\eta[\![H^*]\!]^\sigma)^\partial \subseteq (\eta[\![H^* + \hat{H}]\!]^\sigma)^\partial$.

- Case ($\mathtt{T2{-}Wkn}$). The typing rule is

$$\frac{\Gamma, \bar{H} \vdash^\sharp_g e : \tau \quad \sigma \models g \quad [\![\bar{H}]\!]^\sigma \subseteq [\![H]\!]^\sigma}{\Gamma, H \vdash^\sharp_g e : \tau}$$

By applying the inductive hypothesis to $\Gamma, \bar{H} \vdash^\sharp_g e : \tau$ we know that, for all g' s.t. $g' \Rightarrow g$, there exists $\bar{H}'$ such that $\Gamma, \bar{H}' \vdash^\sharp_{g'} e' : \tau$ and $(\eta'[\![\bar{H}']\!]^\sigma)^\partial \subseteq (\eta[\![\bar{H}]\!]^\sigma)^\partial$. We conclude observing that $(\eta[\![\bar{H}]\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$ and applying the transitivity of $\subseteq$.

- Case ($\mathtt{T2{-}Str}$). The typing rule is

$$\frac{\Gamma, H \vdash^\sharp_g e : \tau \quad g \Rightarrow g'}{\Gamma, g'H \vdash^\sharp_g e : \tau}$$

and we have that $\eta, e \rightarrow_\pi \eta', e'$. We apply the inductive hypothesis and we obtain that for all $\bar{g}$ such that $\bar{g} \Rightarrow g$ then $\Gamma, \bar{H} \vdash^\sharp_{\bar{g}} e' : \tau$ implies that $\forall \sigma.\sigma \models \bar{g} \implies (\eta'[\![\bar{H}]\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$. We conclude observing that, by definition, $\sigma \models \bar{g}$ implies $\sigma \models g'$ and, then, $[\![g'H]\!]^\sigma = [\![H]\!]^\sigma$.

□

Lemma 5.2 (Subject reduction) Let $\Gamma, H \vdash_g e : \tau$ and $\eta, e \rightarrow^*_\pi \eta', e'$. For each g' such that $g' \Rightarrow g$, there exists H' such that $\Gamma, H' \vdash_{g'} e' : \tau$ and $\forall \sigma.\sigma \models g' \implies (\eta'[\![H']\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$ □

Proof. We proceed by induction on the length of the derivation.

- Base case. In this case $e = e'$ and $\eta = \eta'$. Then the property is trivially satisfied by lemma 5.1.
- Inductive step. We have $\eta, e \rightarrow^*_\pi \eta', e' \rightarrow_\pi \eta'', e''$. We apply the inductive hypothesis to η', e'. Then, applying property 5.1 we have that $\Gamma, H'' \vdash_{g''} e'' : \tau \implies \Gamma, H'' \vdash^\sharp_{g''} e'' : \tau$. Now we apply property 5.3 and we obtain

  $$(\eta''[\![H'']\!]^\sigma)^\partial \subseteq (\eta'[\![H']\!]^\sigma)^\partial \subseteq (\eta[\![H]\!]^\sigma)^\partial$$

  That is the thesis.

□

Theorem 5.1 (Type safety) If $\Gamma, H \vdash_{true} e : \tau$ and $\varepsilon, e \rightarrow^*_\pi \eta', v$, then $\forall\ \sigma.\ \exists\ \eta \in [\![H]\!]^\sigma$ such that $\eta = (\eta')^\partial$. □

Proof. By lemma 5.2 (and by noticing that $\forall \sigma, H.[\![\varepsilon]\!]^\sigma \subseteq [\![H]\!]^\sigma$) we know that for each g such that $g \Rightarrow true$ then $\forall \sigma.\sigma \models g \implies (\eta[\![\varepsilon]\!]^\sigma)^\partial \subseteq ([\![H]\!]^\sigma)^\partial$. The thesis follows from property 5.2. □

Theorem 5.2 Given two plans π and π', if $\Gamma, H \vdash_g e\ |_\pi : \tau \quad \text{and} \quad \Gamma, H' \vdash_g e\ |_{\pi;\pi'} : \tau$ then $\forall \sigma.[\![H']\!]^\sigma \subseteq [\![H]\!]^\sigma$ □

Proof. By induction on the structure of e. The only interesting case is $e = \mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$ (the others are trivial or direct consequences of the inductive hypothesis). According to the definition of plan three cases arise:

- $\pi(\rho) = \ell$. Then $e\ |_\pi = e\ |_{\pi;\pi'} = e'$ and we conclude by applying the inductive hypothesis to e'.
- $\pi(\rho) = \bot$ and $\pi'(\rho) = \ell$. If $e\ |_{\pi;\pi'} = e\ |_\pi = e$ the property is trivially satisfied. Otherwise, $H = \sum_{i \in I} H_i$ and $H' = H_j$ for some $j \in I$. Hence $\forall \sigma.[\![H']\!]^\sigma \subseteq [\![H]\!]^\sigma$.
- $\pi(\rho) = \pi'(\rho) = \bot$. In this case we have $H = H'$.

□

Property 5.4 For all e and π such that $e\,|_\pi$ is complete, $\forall\,\pi'\,.\,e\,|_{\pi;\pi'} = e\,|_{\pi';\pi} = e\,|_\pi$.

Proof. By induction on the structure of e. Again, the only non trivial case is for $e = \mathtt{req}_\rho\,\tau \xrightarrow{\varphi} \tau'$. However, by definition of modular plans, we know that $\pi(\rho) = \pi;\pi'(\rho) = \pi';\pi(\rho) = e_\ell$ and we conclude by applying the inductive hypothesis to e_ℓ. □

Lemma 5.3 Given two terms e,e' and two modular plans π,π' complete for e and e', respectively, then $\pi;\pi'$ is complete for both e and e'. □

Proof. A direct consequence of property 5.4. □

Theorem 5.3 Let π_i be modular plans complete for services e_i, and let $\emptyset, H_i \vdash_{g_i} e_i : \tau_i$, $i = 0, 1$. Then,

- $\pi_0;\pi_1$ is complete for both $\alpha(e_0)$ and $\mathtt{if}\,g\,\mathtt{then}\,e_0\,\mathtt{else}\,e_1$
- $\forall\varphi$, if $H_0 \models \varphi$ then
  - $\pi_0;\pi_1$ is complete for $\varphi[e_0]$
  - $\{\rho \mapsto \ell\};\pi_0$ is complete for $\mathtt{req}_\rho\,\tau \xrightarrow{\varphi} \tau'$, with $e_\ell = e_0$ and $e_\ell : \tau \xrightarrow{H_0} \tau' \in \mathtt{Srv}$
- if $\tau_0 = \tau_1 \xrightarrow{H_0'} \tau'$ and $H_0 \cdot H_1 \cdot H_0'$ is valid, then $\pi_0;\pi_1$ is complete for $e_0 e_1$

□

Proof. All the cases are a consequence of the definition of complete plan and lemma 5.3. □

Property 5.5 For each usage automaton $A_\psi = \langle \mathsf{Ev}, Q, \imath, F, T\rangle$ holds that

$$\forall\,\eta\,:\,\exists\,q \in Q\,:\,\imath \xrightarrow{\eta}{}^* q$$

Proof. A direct consequence of the definition of usage automaton. □

Property 5.6 Let $\mathtt{Preq}(H, A_\psi) = A_{\psi^\star}$ then

$$\forall\,\eta\,:\,\eta \models \psi^\star \Rightarrow \nexists\,q \in Q^\star\,:\,q \xrightarrow{\eta^R}{}^* q_f \wedge q_f \in F'$$

Proof. By contradiction. Let assume that there exists some η such that

(1) $\eta \models \psi^\star$ and

(2) $\exists\,q \in Q^\star\,:\,q \xrightarrow{\eta^R}{}^* q_f \wedge q_f \in F'$

We observe that (2) holds if and only if

$$\eta^R \in \mathcal{L}(N_{\psi^\star}) \Leftrightarrow \eta \in \mathcal{L}(N^R_{\psi^\star}) \Leftrightarrow \eta \in \mathcal{L}(A_{\psi^\star})$$

(by definition of operator R and by property of the NFAs). Hence, we conclude by verifying that $\eta \in \mathcal{L}(A_{\psi^\star}) \Leftrightarrow \eta \not\models \psi^\star$ (by definition of usage automaton), contradicting (1). □

Property 5.7 Let $\texttt{Preq}(H, A_\psi) = A_{\psi^\star}$ then

$$\forall\, \eta \,:\, \exists\, \sigma \,:\, \eta \in [\![H]\!]^\sigma \Rightarrow \exists\, q^\star \in Q^\star \,:\, \imath' \xrightarrow{\eta^R}{}^* q^\star$$

Proof. By property 5.5 there exists $q \in Q'$ such that $\imath' \xrightarrow{\eta^R}{}^* q$ is a valid sequence of transitions for A^R_ψ. Hence, $\eta^R \in \mathcal{L}(A_q)$ and $\eta^R \in [\![H^R]\!]^\sigma$. □

Theorem 5.4 Let $\texttt{Preq}(H, A_\psi) = A_{\psi^\star}$ then

$$\forall\, \eta \,:\, \eta \models \psi^\star \Rightarrow \eta H \models \psi$$

□

Proof. By contradiction. Let assume that there exists some η such that

(1) $\eta \models \psi^\star$ and

(2) $\eta \cdot H \not\models \psi$

We note that (2) holds if and only if

$$\exists\, \sigma \,:\, [\![\eta \cdot H]\!]^\sigma \cap \mathcal{L}(A_\psi) \neq \emptyset \Leftrightarrow \exists\, \bar{\eta} \in [\![H]\!]^\sigma \,:\, \eta\bar{\eta} \in \mathcal{L}(A_\psi) \Leftrightarrow \bar{\eta}^R\eta^R \in \mathcal{L}(A^R_\psi) \qquad (\diamond)$$

Then, applying property 5.7 we obtain that

$$\exists\, q^\star \in Q^\star \,:\, \imath' \xrightarrow{\bar{\eta}^R}{}^* q^\star$$

However, by property 5.6 we know that

$$\nexists \bar{q} \in Q^\star \,:\, \bar{q} \xrightarrow{\eta^R}{}^* q_f \wedge q_f \in F'$$

Hence, we obtain

$$\bar{\eta}^R\eta^R \notin \mathcal{L}(A^R_\psi)$$

and we find a contradiction with $(\diamond)$. □