UNIVERSITÀ DEGLI STUDI DI SIENA
DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE
"R. MAGARI"
DOTTORATO DI RICERCA IN LOGICA MATEMATICA E
INFORMATICA TEORICA (LOMIT)

PH.D. THESIS

# Synthesis of Secure Systems

Ilaria Matteucci

SUPERVISOR

Dr. Fabio Martinelli

April 2008

Dipartimento di Scienze Matematiche e Informatiche "R. Magari",
Pian Dei Mantellini 44, I-53100 Siena
E-mail: ilaria.matteucci@iit.cnr.it

# Abstract

This thesis concerns the study, the development and the synthesis of mechanisms for ensuring the security of complex systems, *i.e.*, systems composed by several interactive components.

A complex system under analysis is described as an *open* system, in which a certain component has an unspecified behavior (not fixed in advance). Regardless of the unspecified behavior, the system should work properly, *e.g.*, should satisfy a certain property. Within this formal approach, we propose techniques to enforce properties and synthesize controller programs able to guarantee that, for all possible behavior of an unspecified agent, the system results secure. In particular, we focus on this last aspect. We look for the existence of an implementation that, by monitoring the behavior of the unspecified component, guarantees the fulfillment of some secure requirements.

We give necessary and sufficient conditions on the behavior of this unspecified component. Then, we describe run-time enforcement mechanisms to force its behavior, as prescribed by the conditions. Hence, we automatically synthesize controller programs.

We contribute within the area of the enforcement of security properties by proposing a parametric and automated framework that goes beyond the definition of how a system should behave to work properly. Indeed, while the majority of related work focuses on the definition of monitoring mechanisms, we do not only present enforcing techniques, but we also aid in their synthesis. The presented approach can be applied to different scenarios. For instance, we define controller operators to enforce security properties for system in a timed setting and we deal with parameterized systems. As a further result, we present a tool for the synthesis of secure systems.

# Acknowledgments

I wish to thank all the people that helped me during my PhD course.

Firstly, I would like to thank my supervisor Fabio Martinelli, who encouraged and inspired me to do even better. He introduced me to the world of formal methods for security analysis and he followed my research.

I am immensely grateful to everybody who has supported me and put up with me every day. I am indebted to all the people that, for a short or a long period have shared B64 and Lab 18 with me, in particular Lucia Ghelardi who has been very important during the writing of this thesis. Furthermore, I would like to thank my PhD mates and the security section of the Institute of Informatics and Telematics of CNR. A special thanks to Marinella Petrocchi for her patience. Her support has been fundamental for my work as well as for my life.

I am also indebted to my external referees Pierpaolo Degano and Sandro Etalle who have accepted to read my thesis.

I wish to thank all my friends for being close to me even in the worst period of my life and that helped me to overcome it. In particular Irene Ducci e Veronica Chichizzola.

A particular thanks to my family, especially to my mother to whom this thesis is not dedicated to even if she has always been my guide.

Finally, a big thank you to Gianluca who gave me the much needed tranquillity to complete this work.

*to my father Renzo*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The term "*security*" refers to something that provides freedom from danger or anxiety, *i.e.*, it provides safety. This is an abstract concept. However, whenever it is accompanied with terms as "computer" or "information", the term *security* means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification or destruction.

Security has to do with *confidentiality*, *integrity* and *availability* ($CIA$ triad) of electronic information that is processed by or stored on computer systems. In particular:

**Confidentiality** is assurance of data privacy. Only the intended and authorized recipients, *e.g.*, individuals, processes or devices, may read the data. Disclosure to unauthorized entities, for example using unauthorized network sniffing, is a confidentiality violation.

**Integrity** is assurance of data non-alteration. Data integrity means ensuring that the information has not been altered in transmission, from origin to reception. Source integrity is the assurance that the sender of that information is who it is supposed to be. Data integrity can be compromised when information has been corrupted or altered, willfully or accidentally, before it is read by its intended recipient. Source integrity is compromised when an agent spoofs the source identity and supplies incorrect information to a recipient.

**Availability** is assurance in the timely and reliable access to data services for authorized users. It ensures that information or resources are available when required. Most often this means that the resources are available at a rate which is fast enough for the wider system to perform its task as intended. It is certainly possible that confidentiality and integrity are protected, but an attacker causes resources to become less available than required, or not available at all, by actuating a so called Denial of Service attack ($DoS$).

Moreover, security has to do also with *non-repudiation*, that is the concept of ensuring that a contract cannot be denied by either of the parties involved, *authenticity* and *information*

*flow*, which aims at controlling the way the information may flow among different entities, and even more, depending on the application one has in mind.

The diffusion of distributed systems and networks has increased the number of interesting scenarios in which security has a significant role. In particular, in the last few years, the amount of information and sensible data that circulate on the net has been growing up. This is one of the important reasons that have contributed to stimulate research towards security, by studying new techniques for specifying, verifying and synthesizing secure systems.

## 1.1 The goal of this thesis

This thesis concerns the *synthesis of secure systems*.

In this work we consider a *secure system* as a system that satisfies some *security properties* specifying acceptable executions of programs. For example, a security property might concern either *access control*, that specifies what operations individuals can perform on objects, or *information flow*, that specifies what individuals can infer about objects by observing a system behavior, or *availability*, that prohibits to an entity the use of a source, as a result of execution of that source by other entities.

The problem of *synthesis*, first addressed by Merlin and Bochman in [99], occurs when one deals with a system in which there are some unspecified components, *e.g.*, a not completely implemented software. When considering a partially specified system, one may wonder if there exists an implementation that can be plugged into the system, replacing the unspecified one, by satisfying some properties of the whole system. Hence the problem that must be solved is the following one:

$$\exists Y \quad S(Y) \models \phi$$

where $\phi$ is a logic formula representing the property to be satisfied.

The problem of the *synthesis of secure systems* is slightly different. Let us consider a system that we want to be secure. We can study it as a partially specified system. The unspecified part is a component whose behavior is not known a priori, and we want the system to be secure, whatever the behavior of the unspecified components is. With $S$ the system, $X$ the unspecified component, $S(X)$ the partially specified system, we require that:

$$\forall X \quad S(X) \models \phi$$

where, again, $\phi$ is a logic formula representing the property. This formalization has been introduced in [85, 89] as a paradigm to do security analysis by considering that the unspecified component as a possible malicious agent. In this way, requiring that for all $X$ the system $S(X)$ satisfies $\phi$ means requiring that the system is secure regardless whatever the behavior of a possible intruder is.

In the case our requirement does not holds, we wonder if there exists an implementation that, by monitoring the behavior of the unspecified component $X$, guarantees the

system satisfies the required security property, *i.e.*,

$$\exists Y \quad \forall X \quad S(Y \triangleright X) \models \phi$$

where $\triangleright$ is a symbol denoting the fact that $Y$ monitors the behavior of $X$.

In this thesis, *run-time enforcement mechanisms* (or *monitors*) are studied and developed to guarantee the correct behavior of a system. They work by controlling the behavior of possible un-trusted components. These monitors permit to check only possible un-secure part of the system.

Moreover, we do not only define monitors and enforcement mechanisms but also we generate an implementation able to guarantee that the analyzed system results secure, *i.e.*, we definitely obtain an implementation $Y$ that solves the problem of secure system synthesis.

## 1.2 Formal methods

In computer science and software engineering, formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems. The use of formal methods is motivated by the expectation that, by performing appropriate mathematical analysis, system design may be proved reliable and robust.

### Process Algebras

A *process* is a series of actions or events, and an *algebra* is a calculus of symbols combined according to certain defined laws.

A *process algebra* is a formal description technique for complex computer systems, especially those with communicating, concurrently executing components. A number of different process algebras have been developed, *e.g.*, the *Algebra of Communicating Processes* ($ACP$, see [23]), the *Calculus of Communicating Systems* ($CCS$, see [103]), the *Theoretical Communicating Sequential Processes* ($TCSP$, see [28]) and the $\pi$-calculus (see [104]), *et al.* and they all share the following key ingredients:

**Compositional modeling.** Process algebras provide a small number of constructs for building larger systems up from smaller ones. $CCS$, for example, contains only six operators, including one for composition of systems in parallel and others for choice and scoping.

**Operational semantics.** Process algebras are typically equipped with Plotkin-style (see [111] and Section 2.3.1) structural operational semantics ($SOS$) that describes the single-step execution capabilities of a system. Using $SOS$, systems represented as terms in the algebra can be "compiled" into *Labeled Transition Systems* (see Section 2.2.1).

**Behavioral reasoning via equivalences and preorders.** Process algebras also feature the use of *behavioral relation* (see Section 2.3.3) as a mean for relating systems. These relations are usually equivalences, which capture a notion of "same behavior", or preorders, which capture notions of "refinement".

We use process algebras to study security aspects of systems that are specified as processes. Processes are obtained by exploiting operators of the algebras. In particular, in this thesis, we mainly concentrate on $CCS$. In this calculus, the notion of communication between processes is crucial. In fact, the main operator is the parallel one, denoted by $\|$, that permits two processes to communicate by performing complementary actions, *e.g.*, send-receive operations.

## Compositional Analysis

Compositional analysis techniques have been developed for many concurrent languages (see, *e.g.*, [2, 62, 77, 79, 134]). These techniques are based on the structure of the processes. Compositional analysis consists in applying a sequence of *reductions*, each of them transforming a satisfaction problem of a composite process, *i.e.*, deciding whether a given model $\mathcal{M}$ *satisfies* a given assertion $\phi$ (in symbols, $\mathcal{M} \models \phi$), into an equivalent satisfaction problem for an intermediate subcomponent.

For instance, let $A$ and $B$ be two processes and let $\|$ be the parallel operator of the $CCS$ process algebra. Let we suppose to require that the system $A\|B$ satisfies a logic formula $\phi$. A typical rule for compositional reasoning is the following:

$$\frac{A \models \phi_1 \; B \models \phi_2}{A\|B \models \phi}$$

In this way, the problem to check whether the composed process $A\|B$ satisfies $\phi$ is reduced to check whether process $A$ satisfies $\phi_1$ and process $B$ satisfies $\phi_2$, where $\phi_1$ and $\phi_2$ are two logic formulas in which $\phi$ could be decomposed, *i.e.*, $\phi_1 \circ \phi_2 \Rightarrow \phi$ where $\circ$ is a certain composition operator. The choice of the right decomposition of the property $\phi$ in the two sub-properties is usually a difficult task.

In [77], Larsen and Xinxin tackle a problem related to the correct decomposition of properties (see Section 2.4.2). They study how to compute the properties that unspecified subcomponents of a system must satisfy in order to obtain that the whole system satisfies a certain requirement. Moreover, they address the problem of finding properties of subcomponents that are as *weak* as possible, in order not to unnecessarily restrict the choice of further implementation steps. In this way, after defining a partial implementation, it is possible to find the minimal set of properties that the unspecified components must ensure in order to build a complete system with particular requirements.

Also Andersen in [2, 3] presents compositional approach to verifying whether processes satisfy assertions from the logic where processes are drawn from a process language. He describe the *partial model checking* techniques. The method is compositional

in the structure of the processes and works purely on the syntax of processes. It consists in applying a sequence of reductions. In this way, given a system with undefined subcomponents it is possible to find the necessary and sufficient conditions that these subcomponents must satisfied to be sure that the system works properly.

## Open systems for security analysis

A system is said to be *open* if it has some unspecified components. The analysis of an open system consists in studying the behavior of the whole system with respect to all the possible behavior of the unspecified components.

*An open system satisfies a property if and only if, whatever a specified component is substituted to the unspecified one, the whole system satisfies this property.*

According to the approach proposed in [85, 86, 89], it is possible to use the open system paradigm to do security analysis by considering the unspecified components of the system as potential attackers.

Several situations, which commonly arise in computer security analysis, may be regarded as instances of open systems verification, *e.g.*, :

- Security protocols involve several parties sending and receiving information over a possibly insecure network. We can then imagine a hostile intruder being "present into" the network and being able to listen, tap into and fake messages to attack the protocol. We can also imagine that some of the "legitimate" parties start to behave maliciously, trying to achieve an advantage for themselves. These two different scenarios can both be modelled by open systems.

  - In the first scenario, in the system there are, *e.g.*, two parties $A$ and $B$ that are communicating. This situation is usually described through the term $A \| B$ (see Section 2.3.2). If we want to consider a possible "listener", it would be better to consider instead the open system $A \| B \| (\_)$, where the unknown component (the hole) may be used to take into account the presence of the listener, whose behavior we are not able to predict. Note that $A$ and $B$ are not necessarily aware of the presence of the intruder.
  - In the second scenario, there is a party that does not behave as declared. Let $B$ be this party. The situation should be properly modeled by analyzing the context $A \| (\_)$. Here, we remove the information about the "intended" behavior of a certain user, *i.e.*, $B$. However, $A$ still assumes the presence of $B$ with such "intended" behavior.

- Suppose to be the owner of a mobile phone and to have downloaded a tool for it. If the origin of this tool is unknown, one cannot know about the correct functioning of the phone, upon the tool installation. We can model this scenario by considering the tool as the hole in the system, *i.e.*, as the unspecified component.

Thus, when analyzing security-sensitive systems, neither the enemy's behavior nor the malicious users' behavior should be fixed beforehand. This will prevent us from making unjustified assumptions which could lead to erroneous (and dangerous) verification results. To sum up, a system should be secure regardless of whatever behavior the malicious users or intruders may have. This is what we want to guarantee in this thesis, by developing enforcement mechanisms able to control the behavior of the possible malicious components and to prevent the system to be unsecure.

## 1.3 Our main line of research

Given a system $S$ and a security property expressed by a logic formula $\phi$ we want to guarantee that $S$ is secure, *i.e.*, $S$ satisfies the formula $\phi$, against whatever possible intruder or malicious user, hereafter denoted by $X$.

Following the open system approach, we study a system composed by a known part $S$ and an unspecified component $X$. They operate together. Let $\phi$ be a logic formula that expresses a security policy. The verification goal is to check

$$\forall X \quad S \| X \models \phi$$

Since it is not always possible to check all possible behavior of the component $X$, we develop mechanisms that monitor the behavior of the component $X$ and eventually force it in order to guarantee the system works properly. First of all, we apply the *partial model checking* function (see [2, 3] and Section 2.6) to the above equation, in order to evaluate the formula $\phi$ by the behavior of $S$. In this way we obtain a new formula $\phi' = \phi_{//S}$ and we have to monitor only the un-trusted part of the system, here $X$. Thus, we study whether a potential attacker exists and, in particular, which are the necessary and sufficient conditions that this enemy should satisfy for altering the correct behavior of the system. Hence, in order to force $X$ to behave correctly, *i.e.*, as prescribed by $\phi'$, we define controller operators, denoted $Y \triangleright X$.

Our approach allows to automatically synthesize a controller program $Y$ for a given controller operator $Y \triangleright X$, by exploiting satisfiability procedures for temporal logic.

To sum up, in this thesis we define controller operators and present a technique to guarantee that a system is secure by, whatever the behavior of a possible intruder is, allowing the system to behave as prescribed by its specification. In particular we show a method to synthesize a controller program $Y$ for a specified controller operator $\triangleright$.

### Contribution

In this thesis we propose a general framework to enforce security properties that is able to deal with several problems. By using this technique, we can treat systems in a timed setting, parameterized systems, distributed systems and composition of properties. Some application fields of our work are mobile phones and web services.

Another advantage of our approach for enforcing is that we are able to control only the possible un-trusted component of a given system. Other approaches deal with the problem of monitoring the possible un-trusted component to enjoy a given property, by treating it as the whole system of interest. However, it is frequent the case where not the whole system needs to be checked (or it is simply not convenient to check it as a whole). Some components could be trusted and one would like to have a method to constrain only un-trusted ones (*e.g.*, downloaded applets). Similarly, it could not be possible to build a monitor for a whole distributed architecture, while it could be possible to have it for some of its components.

We contribute by showing how our approach could also be used to deal with web services. In particular, in web services scenarios we will be able to synthesize an orchestrator process that, by managing the service providers, guarantees that a given user's requirement is satisfied.

Finally, we have developed a tool that permits to generate a controller program for a specified controller operator. As a matter of fact, we have implemented a synthesis tool in the objective language O'caml [78] that, given a system, a security property, and a controller operator enforcing that property, is able to generate the respective controller program.

## Bibliographical note

The work we present here has been already published in international conferences. In particular:

- The modeling and synthesis of controller operators for safety properties appeared in [91, 92, 93].
- The synthesis of controller operators for information flow properties in [90, 98].
- The synthesis of web services orchestration in [95].
- The synthesis of controller operators for security properties in distributed systems in [94].
- The synthesis tool in [97].

We acknowledge joint research with Fabio Martinelli. In addition to the contribution of University of Siena, Italy, this research work has been done with the contribution of the Institute of Informatics and Telematics, CNR, Italy. We also thank the European projects S3MS, GRIDTRUST and SENSORIA for having partially supported this research.

## Organization of the thesis

The thesis is organized in six chapters and one appendix. In particular,

**Chapter 2** introduces some general background, aimed at helping the reader in acquiring the basic notions required for the subsequent chapters. More precisely, general notions about logic and the definition of syntax and semantics of *linear* and *branching time* logic will be given. Secondly, the definition of process algebras, in particular $CCS$ process algebra, and the definition of behavioral equivalences between processes, will be presented. Finally, the compositional analysis method is recalled.

**Chapter 3** introduces the notion of *enforcement mechanisms* and shows the controller operators that we have developed in order to deal with different security properties. Indeed, different operators can be used to enforce different security properties. Moreover, the chapter introduces an enforcement mechanism on traces based on partial model checking and ideas of *online model checking* and *model checking a path*.

**Chapter 4** presents the synthesis of controller programs for controller operators and it shows possible applications of our framework, within timed setting, and for parameterized or distributed systems. We also treat the problem of composition of properties.

**Chapter 5** shows the tool architecture and some applications.

**Chapter 6** summarizes contents and results of the thesis.

In the Appendix there are the proofs of the results that we present throughout the work, in particular in Chapters 3 and 4.

# Chapter 2

# Logic, process algebra and partial model checking

In this chapter, we start by recalling some basic notions about logic, in particular, by presenting *first order logic*. Then, we present definition of temporal logics as *linear temporal logic* and $\mu$-calculus as *branching time logic*. In particular we focus our attention on some variants of the $\mu$-calculus: the *modal $\mu$-calculus*, the *simultaneous fixpoint $\mu$-calculus* and the *equational $\mu$-calculus*.

Successively, we introduce the *Calculus of Communicating Systems* ($CCS$ for short, see [101, 102]), by giving the operational semantics of programs in the style proposed by Plotkin (see [111]), namely *Structured Operational Semantics* ($SOS$). This language belongs to the family of *process algebras* (see [65, 67]), that are formalisms for the description of concurrent communicating processes.

Finally, we show the compositional analysis techniques proposed by Larsen and XinXin in [77]. They introduced the concept of *context* to deal with partially specified systems. We show how $CCS$ operators can be seen as contexts. We also recall the work of Andersen (see [2, 3, 77]) that analyzed a similar problem by introducing the *partial model checking* technique.

## 2.1 Logical basic notions

In this section, we recall some basic notions about logic. We present *propositional logic* and *first order logic*, $FOL$ for short, that is a formal deductive system used by mathematicians, philosophers, linguists, and computer scientists. For this part we refer to the definition given in [57].

### 2.1.1 Propositional Logic

A *proposition* is an assertion that can be true or false, but it cannot be true and false at the same time.

In logic and mathematics, a *propositional calculus* (or a *sentential calculus*) is a formal system in which formulas representing propositions can be formed by combining atomic propositions using *logical connectives*, and a system of formal proof rules allows certain formulas to be established as "theorems" of the formal system.

The language of a propositional calculus consists of:

- a set of primitive symbols, variously referred to as atomic formulas, placeholders, proposition letters, or variables;
- a set of operator symbols, variously interpreted as logical operators or logical connectives.

In the mathematical formalization, logical connectives are denoted as $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implication) and $\leftrightarrow$ (double implication). Their meaning can be found in Table 2.1.

| |
|---|
| $\neg p$ means "not $p$" |
| $p \wedge q$ means "$p$ and $q$" |
| $p \vee q$ means "$p$ or $q$" |
| $p \rightarrow q$ means "if $p$ then $q$" |
| $p \leftrightarrow q$ means "$p$ if and only if $q$" |

Table 2.1: Logical connectives.

In general, a calculus is a formal system that consists of a set of syntactic expressions, a distinguished subset of these expressions, plus a set of formal rules that define a specific binary relation, intended to be interpreted as logical equivalence, on the space of expressions.

Within a formal, logical system, expressions are mathematical statements, and rules, known as *inference rules*, are typically intended to be truth-preserving. Rules may include *axioms*, *i.e.*, sentences or propositions that are not proved nor demonstrated and that are considered as self-evident. They can then be used to derive ("infer") formulas.

The set of axioms may be empty, a nonempty finite set, a countably infinite set, or can be given by axiom schemata. A formal grammar recursively defines the expressions and well-formed formulas of the language. In particular, a well-formed formula is any atomic formula or any formula that can be built up from atomic formulas by means of operator symbols according to the grammar rules.

The semantics of the propositional calculus assigns a truth function to each proposition. First, it is necessary to define the meaning of the logical connectives. The set of *truth values* is the set $\{\mathbf{T}, \mathbf{F}\}$. *Truth tables* show the meaning of the boolean connectives as follows.

The first two columns represent the four possible truth values of $A$ and $B$, respectively. The other columns show truth values of the propositions $A \wedge B$, $A \vee B$, $A \rightarrow B$ and $A \leftrightarrow B$.

| $A$ | $B$ | $A \wedge B$ | $A \vee B$ | $A \rightarrow B$ | $A \leftrightarrow B$ |
|---|---|---|---|---|---|
| **F** | **F** | **F** | **F** | **T** | **T** |
| **F** | **T** | **F** | **T** | **T** | **F** |
| **T** | **F** | **F** | **T** | **F** | **F** |
| **T** | **T** | **T** | **T** | **T** | **T** |

Table 2.2: Truth table for boolean connectives.

### 2.1.2 First Order Logic

First Order Logic $FOL$ is a system of deduction extending propositional logic by allowing quantification over individuals of a given domain (universe) of discourse. For example, the following proposition can be stated: "Every individual has the property $P$". Thus, while propositional logic deals with simple declarative propositions, first-order logic additionally covers predicates and quantification. Let us consider the following sentences: "Socrates is a man", "Plato is a man". In propositional logic these will be two unrelated propositions, denoted, for example, by $p$ and $q$. In first-order logic they can be connected by the same relation: $Man(x)$, where $Man(x)$ means that "$x$ is a man". When $x = Socrates$ we get the first proposition, *i.e.*, $p$, and when $x = Plato$ we get $q$. We obtain a more powerful logic than the propositional logic. Indeed, there is the possibility to build constructions with quantifiers, *e.g.*, "for every $x$, if $Man(x)$, then ...". However, without quantifiers, every valid argument in $FOL$ is valid in propositional logic, and viceversa.

The language of $FOL$ has sufficient expressive power for the formalization of most of mathematics. A first-order theory consists of a set of axioms (usually finite or recursively enumerable) and the statements deducible from them.

Basically, two extra components are present in $FOL$, with respect to propositional logic. Quantifiers, introduced above, and variables, referring to properties. Quantifiers govern the nature of the quantity of the variables bound within their scope. There are two types of quantifiers - *universal* and *existential*, to be read "*for all* $x$" and "*there exists at least one* $x$", respectively.

#### Syntax

The *signature* of a first order logic consists of a set of symbols divided in three subsets, *constant symbols*, *function symbols* and *relation symbols* and a function, called *arity* that associates to constant symbols the value zero and to function symbols positive integers. There is also a set, denoted by $Var$, of *variables* symbols. We can give the following definition of *terms*.

**Definition 2.1** *The set of terms is recursively defined by the following rules:*

- *Any constant is a term.*
- *Any variable is a term.*

- *if $t_1, \ldots, t_n$ are terms and $f$ is a function symbols of arity $n$ then $f(t_1, \ldots, t_n)$ is a term.*
- ***Closure clause**: Nothing else is a term.*

A *closed* term is a term with no variables.

**Definition 2.2** *A well-formed* formula *is recursively defined as follows.*

- *If $R$ is a symbol of relation of arity $n$ and $t_1, \ldots t_n$ are terms then $R(t_1, \ldots, t_n)$ is a formula.*
- *Let $\phi$ and $\psi$ be two formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$ and $\phi \rightarrow \psi$ are formulas.*
- *If $\phi$ is a formula and $x$ is variable then $\forall x\phi$ and $\exists x\phi$ are formulas.*
- ***Closure clause**: Nothing else is a formula.*[1]

A formula is *atomic* if there are no connectives.

A formula $\phi$ can contain variables. If the variable occurs in a subformula in which there is a quantifier then it is called a *bound* variable. If it is not bound, we call it *free*.

### Semantics

First, we recall the following definition.

**Definition 2.3** *Let $L$ be a signature. A $L$-*structure *$M$ consists in:*

- *A non empty set, said* universe*, for short $dom(M)$.*
- *A correspondence $c \mapsto c_M$ between the constant symbols of the signature $L$ and the constant symbols of the structure $M$.*
- *A correspondence $f \mapsto f_M$ between the function symbols of the signature $L$ and the function symbols of the structure $M$.*
- *A correspondence $R \mapsto R_M$ between the relation symbols of the signature $L$ and the relation symbols of the structure $M$.*

For the sake of readability, in the following we will omit to specify the signature with respect to which we give all the definitions.

Let $M$ be a structure and let $\phi$ be a formula with free variables in $\{x_1, \ldots, x_n\}$. The formula $\forall x_1 \ldots x_n\ \phi$ is the *universal closure* of $\phi$.

**Definition 2.4** *Given a formula $\phi$ and a structure $M$ if there exists the universal closure of $\phi$ that is true in $M$, we said that $\phi$ is* valid*, or universally true, in $M$.*

[1] We do not consider languages with the symbol $=$.

**Definition 2.5** *A* theory $T$ *consists in a signature* $L$ *and a set of* axioms *of* $T$.

**Definition 2.6** *A* model *of a theory* $T$ *is a structure in which all axioms of* $T$ *are valid. If* $M$ *is a model of* $T$ *we write* $M \models T$. *This means that for all axioms* $\phi$ *of* $T$, $M \models \phi$. *A theory is said* coherent *or* satisfiable *if it has at least a model.*

## 2.2 Modal Logic

Here, we consider *Modal logic*, characterized by *modal operators* describing, informally, concepts like: "it is necessary that . . .", or "it is possible that . . .".

### 2.2.1 Structures: *Kripke Structures* and *Labeled Transition Systems*

As anticipated in Section 2.1, to establish if a formula is true or false, we should define a structure with respect to which the formula is interpreted.

We recall the definition of two kind of structures that are suitable for our later purposes: *Labeled Transition Systems*, $LTS$ for short, and *Kripke Structures*. They are very similar. In $LTS$ transitions are labeled to describe the actions which cause a change in the state, while in a Kripke Structure states are labeled to describe how they are modified by transitions.

#### Kripke Structure

A Kripke Structure was introduced by Saul Kripke in [73] in order to respond to a difficulty with classical quantification theory. In particular, the key point is the possibility to represent the fact that terms may fail within one scenario (that he called *world*), and exist within another one. Indeed, when standard quantifier rules are used, each term refers to something that exists in all the possible worlds. This seems incompatible with our ordinary practice of using terms to refer to things that exist contingently.

A Kripke Structure represents a graph where nodes are the reachable states of the system, and whose edges represent state transitions. A labeling function maps each node to a set of properties that hold in the corresponding state.

**Definition 2.7** *Let* $AP$ *be the set of atomic propositions. A* Kripke Structure $M$ *over* $AP$ *is a tuple* $M = (S, S_0, R, \delta)$ *where*

- $S$ *is a finite set of states.*
- $S_0 \subseteq S$ *is the set of initial states.*
- $R \subseteq S \times S$ *is a transition relation that must be total, that is, for every state* $s \in S$ *there is a state* $s' \in S$ *such that* $R(s, s')$.
- $\delta : S \rightarrow 2^{AP}$ *is a function that labels such state with the set of atomic propositions that are true in that state.*

When feasible, we omit the set of initial states $S_0$.

**Definition 2.8** *A* path *in the structure $M$ from a state $s$ is an infinite sequence of states $u = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.*

Kripke Structures could be seen as a type of nondeterministic *finite state machine*, that is a model composed by a finite number of states, transitions between these states, and actions. Temporal logics are traditionally interpreted in terms of Kripke Structures (see [45]). As a matter of fact, they are the model used to give semantics (definition of when a specified property holds) for the most widely used specification languages for temporal logics. Kripke Structures describe the behavior of the system in a manner that is independent from the specification language. Therefore, temporal logics are really formalism-independent. The definition of atomic propositions is the only thing that needs to be adjusted according to the formalism. Moreover, they are also used for describing behavior of systems.

### Labeled Transition System

In computer science, *Labeled Transition Systems* ($LTS$s) are commonly used to represent possible computation pathways during the execution of a program. In Kripke Structure the attention is on the formula that is valid in a state. On the other hand, in a $LTS$ the attention is focused on the transition between two states.

$LTS$s consist of a set of states, a set of labels (or *actions*) and a *transition relation*, $\mathcal{T}$, that describes how a process passes from a state to another, *i.e.*, given two states $s$ an $s'$ and a label $a$, $s\mathcal{T}_a s'$ means that the process passes from the state $s$ to the state $s'$ with an arc labeled $a$.

**Definition 2.9** *A triple $\langle S, Act, \mathcal{T} \rangle$ is called* Labeled Transition System *($LTS$), where $S$ is a set of states, $Act$ is a set of actions (or labels) and $\mathcal{T} \subseteq S \times Act \times S$ is a ternary relation, known as a* transition relation.

$LTS$s are mathematical objects used to give formal (operational) semantics to concurrent programming languages (see Section 2.3.1).

### Doubly Labeled Transition System

Some research deals with a third structure that combines both of these aspects. It is called *Doubly labeled transition system* ($L^2TS$) (see [41, 42]).

**Definition 2.10** *Let $AP$ be a fixed set of atomic names. $L^2TS$ is a structure $(S, Act, \mathcal{T}, \delta)$ where $(S, Act, \mathcal{T})$ is a $LTS$ and $\delta : S \to 2^{AP}$ is a labeling function which associates a set of atomic propositions to each state.*

The introduction of this structure permits to compare different temporal logics, *e.g.*, branching and linear temporal formulas. In this way, they can be interpreted on the same structure.

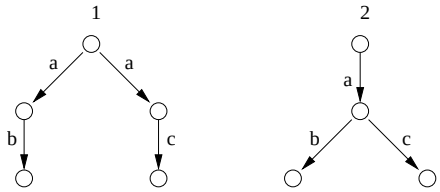



Figure 2.1: The state 2 satisfies $\phi_2$, while the state 1 does not satisfy $\phi_2$.

### 2.2.2 *Hennessy-Milner Logic* and $\mu$-*calculus*

Here, we consider the *Hennessy-Milner Logic*, $HML$ for short. Then, we devote the major part of the section to the $\mu$-calculus by presenting some of its variants: the *modal $\mu$-calculus*, *simultaneous fixpoint $\mu$-calculus* and the *equational $\mu$-calculus*. We choose these three variants because they are the most meaningful for this thesis.

#### *Hennessy-Milner Logic*

$HML$ [102] is a temporal logic well suited for the specification and verification of systems whose behavior is naturally described by changes of states through actions. This is a primitive modal logic. Formulas of $HML$ have, in addition to the Boolean operators, a modality $\langle a\rangle\phi$, where $a \in Act$ and $Act$ is a set of actions (or labels). The meaning is "it is possible to do and $a$-action to go to a state where $\phi$ holds".

$$\phi ::= \mathbf{T} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle a\rangle\phi$$

The basic information are not atomic propositions, that here are only the constants $\mathbf{T}$ and $\mathbf{F}$, but the notion of action or transition between worlds. The semantics of modal logic is given by using Labeled Transition Systems $\langle S, Act, \rightarrow\rangle$, where $\rightarrow$ represents the transition relation, by inductively defining when a process in a transition system satisfies a certain property, *e.g.*, $E \models \langle a\rangle\phi$ if and only if $\exists F$ such that $E$ performs a transition labeled by the action $a$ in order to pass to $F$, *i.e.*, $E \xrightarrow{a} F$, and $F \models \phi$. $E$ and $F$ are processes, and they can be seen as states in a transition system.

The expressive power of $HML$ in this form is quite weak: a given $HML$ formula can only make statements about a given finite number of steps towards the future.

The satisfaction of a formula with respect to a state of an $LTS$ is defined inductively as follows:

$$\begin{array}{lll} s \models \mathbf{T} & & \text{for every } s \\ s \models \neg\phi & \text{iff} & \text{not } s \models \phi \\ s \models \phi_1 \wedge \phi_2 & \text{iff} & s \models \phi_1 \text{ and } s \models \phi_2 \\ s \models \langle a\rangle\phi & \text{iff} & \exists s'(s \xrightarrow{a} s' \text{ and } s' \models \phi) \end{array}$$

**Example 2.1** *We give a simple example that shows how $HML$ formulas may be used to distinguish between two $LTS$s, whose initial states have a different branching structure. Consider the two $LTS$s in Figure 2.1. Let $\phi_1$ be $(\langle a\rangle\langle b\rangle\mathbf{T}) \wedge (\langle a\rangle\langle c\rangle\mathbf{T})$ and let $\phi_2$ be $(\langle a\rangle(\langle b\rangle\mathbf{T} \wedge \langle c\rangle\mathbf{T}))$. Then, state 2 of the rightmost $LTS$ satisfies $\phi_1$ and $\phi_2$, while state 1 of the leftmost $LTS$ satisfies $\phi_1$ but not $\phi_2$.*

### Modal $\mu$-calculus

Modal $\mu$-calculus is a process logic which extends $HML$ with fixpoint operators in order to reason directly about recursive definitions of properties. It permits to analyze non terminating behaviors of systems.

Let $a$ be in $Act$, $Z$ be a variable ranging over a set of variables $V$. Modal $\mu$-calculus formulas are generated by the following grammar:

$$\phi ::= Z \mid \mathbf{T} \mid \mathbf{F} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a\rangle\phi \mid [a]\phi \mid \mu Z.\phi \mid \nu Z.\phi$$

The possibility modality $\langle a\rangle\phi$ expresses the ability to have an $a$ transition to a state that satisfies $\phi$. The necessity modality $[a]\phi$ expresses that after each $a$ transition there is a state that satisfies $\phi$. We consider the usual definition of bounded and free variables. The interpretation of a closed formula $\phi$ with respect to a $LTS$ $M$ is the set of states where $\phi$ is true. The interpretation of a formula $\phi(Z)$ with a free variable $Z$ is a function from set of states to set of states. Hence, the interpretation of $\mu Z.\phi(Z)$ ($\nu Z.\phi(Z)$) is the least (greatest) fixpoint of this function. The interpretation of a formula with free variables is a monotonic function, so a least (greatest) fixpoint exists.

Formally, given an $LTS$ $M = \langle S, Act, \rightarrow\rangle$, where $\rightarrow$ is the transition relation, the semantics of a formula $\phi$ is a subset $[\![\phi]\!]_\rho$ of the states of $M$, defined in Table 2.3, where $\rho$ is a function (called *environment*) from free variables of $\phi$ to subsets of the states of $M$. This kind of semantics is called *denotational semantics*, that is an approach to formalizing the semantics of computer systems by constructing mathematical objects, said *denotations* or *meanings*.

The environment $\rho[S'/Z](Y)$ is equal to $\rho(Y)$ if $Y \neq Z$, otherwise $\rho[S'/Z](Z) = S'$. Actually we have presented a slight variant of the propositional $\mu-$calculus as described by Kozen (see [72]) or by Walukiewicz (see [131]).

In our treatment, we omit propositional symbols. As usual we consider $\phi \Rightarrow \varphi$ as an abbreviation for $\neg\phi \vee \varphi$.

**Examples and facts.** Modal $\mu$-calculus allows to express a lot of interesting properties, like *safety* properties, *i.e.*, nothing bad happens as well as *liveness* properties, *i.e.*, something good happens. Moreover, equivalence conditions over $LTS$s may be expressed through this logic (see [2, 125] and Section 2.2.1).

*Safety* properties are usually defined by means of greatest fixpoint formulas, while *liveness* properties by least fixpoint formulas (see [27]).

$$
\begin{array}{rcl}
[\![T]\!]_\rho & = & S \\
[\![F]\!]_\rho & = & \emptyset \\
[\![Z]\!]_\rho & = & \rho(Z) \\
[\![\phi_1 \wedge \phi_2]\!]_\rho & = & [\![\phi_1]\!]_\rho \cap [\![\phi_2]\!]_\rho \\
[\![\phi_1 \vee \phi_2]\!]_\rho & = & [\![\phi_1]\!]_\rho \cup [\![\phi_2]\!]_\rho \\
[\![\langle a \rangle \phi]\!]_\rho & = & \{s | \exists s' : s \xrightarrow{a} s' \text{ and } s' \in [\![\phi]\!]_\rho\} \\
[\![[a]\phi]\!]_\rho & = & \{s | \forall s' : s \xrightarrow{a} s' \text{ implies } s' \in [\![\phi]\!]_\rho\} \\
[\![\mu Z.\phi]\!]_\rho & = & \bigcap \{S' | [\![\phi]\!]_{\rho[S'/Z]} \subseteq S'\} \\
[\![\nu Z.\phi]\!]_\rho & = & \bigcup \{S' | S' \subseteq [\![\phi]\!]_{\rho[S'/Z]}\}
\end{array}
$$

Table 2.3: Denotational semantics of modal $\mu$-calculus.

An example of *safety* property is a formula that expresses the possibility to open a new file only if the previous one is closed. It can be described by the following formula:

$$\nu Z_1.[\texttt{open}]([\texttt{close}]Z_1 \wedge [\texttt{open}]\mathbf{F})$$

As a matter of fact, this formula states that whenever an action `open` is performed ([`open`]) the process goes in a state in which it is not possible to perform another `open` action ([`open`]$\mathbf{F}$) and it is possible to close the file opened at the previous transition step ([`close`]$Z_1$). It is possible to note that, after performing the action `close`, the process call the first variable $Z_1$. This allows to open a new file after closing the previous one.

Another interesting formula is $\nu Z.[K]Z \wedge [Act \backslash K]\mathbf{F}$[2] which expresses the fact that only actions in $K$ can be performed by a process in any reachable state.

Moreover, *access-control property* are safety properties. The set of proscribed partial executions contains those partial executions ending with an unacceptable operation being attempted. There is no way to "unaccess" the resource and fix the situation afterward.

Also some *bounded availability* properties may be characterized as safety ones. An example is "one principal cannot be denied the use of a resource for more then $D$ steps as a results of the use of that resource by other principals". Here, the defining set of partial executions contains intervals that exceed $D$ steps and during which a principal is denied use of a resource.

A *liveness* property like "there exists a path for a state which satisfies $\phi$" is expressed by $\mu Z.\langle Act \rangle Z \vee \phi$.

Modal $\mu$-calculus can express cyclic properties, *e.g.*, "there exists an infinite path such that the formula $\psi$ is true at all even instants". The $\mu$-calculus formula which expresses the aforementioned property is the following:

$$\nu Z.(\langle Act \rangle \langle Act \rangle Z \wedge \psi).$$

[2]We use an extended notation, with $K \subseteq Act$ let $[K]\phi$ be $\bigwedge_{a \in K}[a]\phi$, and $\langle K \rangle \phi$ be $\bigvee_{a \in K}\langle a \rangle \phi$. Since $Act$ is finite the indexed disjunctions (conjunctions) can be expressed by means of disjunction (conjunction).

The following lemma states some well known facts about $\mu$-calculus (see [33]) that will be used later in the chapter.

**Lemma 2.1 ([33])** *Let $\phi$ be a $\mu$-calculus formula, $\rho$ an environment and $\sigma \in \{\mu, \nu\}$.*

1. *if $Z$ is not free in $\phi$ then $[\![\sigma Z.\phi]\!]_\rho = [\![\phi]\!]_\rho$.*
2. *Let $W$ be a variable that does not appear free in $\sigma Z.\phi$, then $[\![\sigma Z.\phi]\!]_\rho$ $= [\![\sigma W.\phi[W/Z]]\!]_\rho$.*
3. *Let $\psi$ be a formula, then $[\![\phi[\psi/Z]]\!]_\rho = [\![\phi]\!]_{\rho[[\![\psi]\!]_\rho/Z]}$.*
4. *$[\![\sigma Z.\phi]\!]_\rho = [\![\phi[\sigma Z.\phi/Z]]\!]_\rho$.*

Modal $\mu$-calculus subsumes several temporal logics (see [24, 38]). But, despite its expressiveness, the satisfiability problem (namely finding a structure and a state where the formula holds) still remains EXPTIME-complete (see [127]).

Moreover $\mu$-calculus enjoys the *finite model* property, *i.e.*, if a closed formula is satisfiable then there exists a finite model (a finite state process) for that formula (see [127]).

**Theorem 2.1 ([127])** *Given a formula $\phi$, it is possible to decide within exponential time in the length of $\phi$ if there exists a model of $\phi$ and it is also possible to give an example of such model.*

A finitary axiom system has been proposed by Walukiewicz in [131, 132, 133]. We recall here that satisfiability procedure because we refer to it later in this thesis.

**Walukievicz satisfiability procedure.**

**Tableau construction.** The vocabulary of the $\mu$-calculus is extended by a countable set *DCons* of fresh symbols that will be referred to as *definition constant* and usually denoted $U, V, \cdots$ (see [131, 133]). These new symbols are now allowed to appear positively in formulas, like propositional variables. A *definition list* is a finite sequence of equations:

$$\mathcal{D} = ((U_1 = \sigma_1 Z.\phi_1(Z)), \cdots, (U_n = \sigma_n Z.\phi_n(Z)))$$

where $U_1, \cdots, U_n \in DCons$ and $\sigma_i Z.\phi_i(Z)$ are formulas such that all definition constants appearing in $\phi_i$ are among $U_1, \cdots, U_{i-1}$. We assume that $U_i \neq U_j$ and $\phi_i \neq \phi_j$ for $i \neq j$. If $i < j$ then $U_i$ is said to be older than $U_j$.

A *tableau sequent* is a pair $(\Gamma, \mathcal{D})$ where $\mathcal{D}$ is a definition list and $\Gamma$ is a finite set of formulas such that the only constants that occur in them are those from $\mathcal{D}$. We will denote $(\Gamma, \mathcal{D})$ by $\Gamma \vdash_{\mathcal{D}}$.

A *tableau axiom* is a sequent $\Gamma \vdash_{\mathcal{D}}$ such that some formula and its negation occur in $\Gamma$.

In Table 2.4 there is the set $\mathcal{S}$ of rules for constructing *tableau*. In the last rule $(all\langle\rangle)$

| | |
|---|---|
| (and) | $\dfrac{\phi, \varphi, \Gamma \vdash_{\mathcal{D}}}{\phi \wedge \varphi, \Gamma \vdash_{\mathcal{D}}}$ |
| (or) | $\dfrac{\phi, \Gamma \vdash_{\mathcal{D}} \ \varphi, \Gamma \vdash_{\mathcal{D}}}{\phi \vee \varphi, \Gamma \vdash_{\mathcal{D}}}$ |
| (cons) | $\dfrac{\phi U, \Gamma \vdash_{\mathcal{D}}}{U, \Gamma \vdash_{\mathcal{D}}}$ whenever $(U = \sigma Z.\phi(Z)) \in \mathcal{D}$ |
| ($\mu$) | $\dfrac{U, \Gamma \vdash_{\mathcal{D}}}{\mu Z.\phi(Z), \Gamma \vdash_{\mathcal{D}}}$ whenever $(U = \mu Z.\phi(Z)) \in \mathcal{D}$ |
| ($\nu$) | $\dfrac{U, \Gamma \vdash_{\mathcal{D}}}{\nu X.\phi(Z), \Gamma \vdash_{\mathcal{D}}}$ whenever $(U = \nu Z.\phi(Z)) \in \mathcal{D}$ |
| (all $\langle\rangle$) | $\dfrac{\{\phi, \{\varphi : [a]\varphi \in \Gamma\} \vdash_{\mathcal{D}} \quad : \langle a\rangle\phi \in \Gamma\}}{\Gamma \vdash_{\mathcal{D}}}$ |

Table 2.4: System rule $\mathcal{S}$ for constructing a tableau.

each formula in $\Gamma$ is a propositional constant, a variable, a negation of one of them or a formula of the form $\langle b\rangle\varphi$ or $[b]\varphi$ for some action $b$ and a formula $\varphi$.

Observe that each rule, except *(or)* or *(all $\langle\rangle$)*, has exactly one premise.

The system $\mathcal{S}_{mod}$ is obtained from $\mathcal{S}$ by replacing the rule *(or)* by two rules $(or_{left})$ and $(or_{right})$ defined in the obvious way.

The system $\mathcal{S}_{ref}$ is obtained from $\mathcal{S}$ by replacing the rule $(all\langle\rangle)$ by the rule

$$(\langle\rangle) \qquad \frac{\langle a\rangle\phi, \Gamma \vdash_{\mathcal{D}}}{\phi, \{\varphi : [a]\varphi \in \Gamma\} \vdash_{\mathcal{D}}}$$

with the same restrictions on formulas in $\Gamma$ as in the case of $(all\langle\rangle)$ rule.

**Definition 2.11** *Given a positive guarded formula $\phi$, a* tableau *for $\phi$ is any labeled tree $\langle K, L\rangle$, where $K$ is a tree and $L$ a labeling function, such that*

- *the root of $K$ is labeled with $\phi \vdash_{\mathcal{D}}$ where $\mathcal{D}$ is the definition list of $\phi$.*
- *if $L$ is a tableau axiom then $n$ is a leaf of $K$.*
- *if $L(n)$ is not an axiom then the sons of $n$ in $K$ are created and labeled according to the rules of the system $\mathcal{S}$.*

A *quasi-model* of $\phi$ is defined in a similar way to tableau, except the system $\mathcal{S}_{mod}$ is used instead of $\mathcal{S}$ and we impose the additional requirement that no leaf is labeled by a tableau axiom. A *quasi-refutation* of $\phi$ is defined in a similar way to tableau, except the system $\mathcal{S}_{ref}$ is used instead of $\mathcal{S}$ and we impose the additional requirement that every leaf is labeled by a tableau axiom.

**Remark 2.1** *Observe that each quasi-model, as well as a quasi-refutation can be obtained from a tableau by cutting of some nodes.*

Let $\mathcal{P} = (v_1, v_2, \cdots)$ be a path in the tree $K$. A *trace* $\mathcal{T}r$ on the path $\mathcal{P}$ is any sequence of formulas $\{\phi_i\}_{i\in I}$ such that $\phi_i \in L(v_i)$ and $\phi_{i+1}$ is either $\alpha_i$, if formula $\phi_i$ was not reduced by the rule applied in $v_i$, or otherwise $\phi_{i+1}$ is one of the formulas obtained by applying the rule to $\phi_i$.

A constant $U$ *regenerates* on the trace $\mathcal{T}r$ if for some $i$, $a_i = U$ and $a_{i+1} = \alpha(U)$ where $(U = \sigma Z.\phi(Z)) \in \mathcal{D}$. The trace $\mathcal{T}r$ is called a $\nu$-trace if and only if it is finite and does not end with a tableau axiom, or if the oldest constant in the definition list $\mathcal{D}$ which is regenerated infinitely often on $\mathcal{T}r$ is a $\nu$-constant. Otherwise the trace is called a $\mu$-trace.

**Definition 2.12** *A quasi model $\mathcal{PM}$ is called* pre-model *if and only if any trace on any path of $\mathcal{PM}$ is a $\nu$-trace.*

*A quasi-refutation of $\phi$ is called a* refutation *of $\phi$ if and only if on every path of there exists a $\mu$-trace.*

**Game.** Given a formula $\phi$ it is possible to find either a pre-model or a refutation for it. Let $\mathcal{T}$ be a tableau for $\phi$. In order to find a pre-model or a refutation it is supposed to be two players, $I$ and $II$ of a game with the following rules:

- game starts in the root of $\mathcal{T}$.
- in any (*or*) node, *i.e.*, a node where (*or*) rule is applied, player $I$ chooses one of he sons.
- in any ($all\langle\rangle$) node, player $II$ chooses one of the sons.
- in other nodes which are not leaves automatically the only son is chosen.

The result of such a game is either a finite or an infinite path of the tableau $\mathcal{T}$. The path can be finite only when it ends in a leaf which can be labeled either by axiom or by unreducible sequent but not an axiom. In the former case player $II$ wins and in the latter case player $I$ is the winner. If the resulting path is infinite, then player $II$ wins if and only if we can find a $\mu$-trace on the path. A winning strategy of either player can be naturally presented as a tree. More precisely, a winning strategy for player $I$ may be identified with a pre-model of $\phi$ while a winning strategy for player $II$ can be identified with a refutation of $\phi$. Hence, we have the following proposition.

**Proposition 2.1** *For each formula $\phi$ there exists a pre-model or a refutation in any tableau for $\phi$.*

According to [131, 133], a formula $\phi$ is satisfiable if and only if there exists a pre-model for it. In particular here we recall the definition of *canonical structure* that is the candidate to be a model of $\phi$.

**Definition 2.13** *Given a pre-model $\mathcal{PM}$, the* canonical structure *for $\mathcal{PM}$ is a structure $\mathcal{M} = \langle S^{\mathcal{M}}, R^{\mathcal{M}}, \rho^{\mathcal{M}}\rangle$ such that*

- *$S^{\mathcal{M}}$ is the set of all nodes of $\mathcal{PM}$ which are either leaves or to which ($all\langle\rangle$) rule was applied. For any node $n$ of $\mathcal{PM}$ we will denote by $s_n$ the closest descendant of $n$ belonging to $S^{\mathcal{M}}$.*
- *$(s, s') \in R^{\mathcal{M}}(a)$ if and only if there is a son $n$ of $s$ with $s_n = s'$, such that $L(n)$ was obtained from $L(s)$ by reducing a formula of the form $\langle a\rangle\phi$.*
- *$\rho^{\mathcal{M}}(p) = \{s : p$ occurs in the sequent $L(s)\}$.*

**Proposition 2.2 ([131, 133])** *If there exists a pre-model $\mathcal{PM}$ for a positive guarded sentence $\phi$ then $\phi$ is satisfiable in the canonical structure for $\mathcal{PM}$.*

### Simultaneous fixpoint $\mu$-calculus

Another variant of the $\mu$-calculus is the *simultaneous $\mu$-calculus*. This variant permits us to define, in a simultaneous way, recursive properties by allowing minimum and maximum fixed points to be used freely and interchangeably.

We recall here the specification given in [77].

**Definition 2.14** *Let $Act$ be a set of actions and let $V$ be a set of variables. The* formulas *and* declarations *over $V$ relative to $Act$, $\mathcal{F}_{V,Act}$ and $\mathcal{D}_{V,Act}$ are built up according to the following abstract syntax*

$$\phi ::= \mathbf{T} \mid \mathbf{F} \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a\rangle\phi \mid [a]\phi \mid \text{LET MAX } D \text{ IN } \phi \mid \text{LET MIN } D \text{ IN } \phi$$
$$D ::= X_1 = \phi_1 \ldots X_n = \phi_n$$

The above logic is a propositional modal logic with $\langle a\rangle\phi$ and $[a]\phi$ providing the two relativized modalities. The declaration in the LET-constructs, LET MAX $\{X_1 = \phi_1 \ldots\}$ IN $\phi$ and LET MIN $\{X_1 = \phi_1 \ldots\}$ IN $\phi$, introduces simultaneous recursively specified properties $X_1$ with scope being the body $\phi$. The concepts of free and bound variables are defined as usual; in particular we call a formula *closed* if it contains no free variables. We shall use standard notation $\phi[\psi/X]$ to describe the substitution of $\psi$ for all free occurrences of the variable $X$ in $\phi$. The interpretation of the introduced logic is given relative to labeled transition system over the set of action $Act$. A *labeled transition system* is a structure $(\Gamma, L, \rightarrow)$, where $\Gamma$ is a set of *configurations* (or states) and $L$ is a set of labels (or actions) and $\rightarrow \subseteq \Gamma \times L \times \Gamma$ is the *transition relation*. The interpretation of a closed formula is given as the set of configurations *satisfying* the formula. However, as formulas (and declarations) in general may contain free variables, the semantics of the formulas is given with respect to an environment $\rho$. By $\mathcal{E}_{V,Act}$ is defined the set of environments over $V$ (relative to $Act$). The semantics definition is given in Table 2.5 with respect to the following definition of the semantics functions:

$$\mathrm{F} : \mathcal{F}_{V,Act} \rightarrow \mathcal{E}_{V,Act} \rightarrow \mathcal{P}(\Gamma)$$
$$\mathrm{D}_\nu, \mathrm{D}_\mu : \mathcal{D}_{V,Act} \rightarrow \mathcal{E}_{V,Act} \rightarrow \mathcal{E}_{V,Act}$$

$$\mathrm{F}[\![\mathbf{T}]\!]_\rho = \Gamma \quad \mathrm{F}[\![\mathbf{F}]\!]_\rho = \emptyset \quad \mathrm{F}[\![X]\!]_\rho = \rho(X)$$
$$\mathrm{F}[\![\phi_1 \vee \phi_2]\!]_\rho = \mathrm{F}[\![\phi_1]\!]_\rho \cup \mathrm{F}[\![\phi_2]\!]_\rho \quad \mathrm{F}[\![\phi_1 \wedge \phi_2]\!]_\rho = \mathrm{F}[\![\phi_1]\!]_\rho \cap \mathrm{F}[\![\phi_2]\!]_\rho$$
$$\mathrm{F}[\![\langle a \rangle \phi]\!]_\rho = \{\gamma \in \Gamma | \exists \gamma' : \gamma \xrightarrow{a} \gamma' \wedge \gamma' \in \mathrm{F}[\![\phi]\!]_\rho\}$$
$$\mathrm{F}[\![[a]\phi]\!]_\rho = \{\gamma \in \Gamma | \forall \gamma' : \gamma \xrightarrow{a} \gamma' \Rightarrow \gamma' \in \mathrm{F}[\![\phi]\!]_\rho\}$$
$$\mathrm{F}[\![\text{LET MAX} D \in \phi]\!]_\rho = \mathrm{F}[\![\phi]\!](D_\nu[\![D]\!]_\rho) \quad \mathrm{F}[\![\text{LET MIN} D \in \phi]\!]_\rho = \mathrm{F}[\![\phi]\!](D_\mu[\![D]\!]_\rho)$$

and

$$D_\nu[\![X_1 = F_1 \ldots X_n = F_n]\!]_\rho = \nu\rho' \; \rho\{\mathrm{F}[\![\phi_1]\!]_{\rho'} \backslash X_1, \ldots, [\![\phi_n]\!]_{\rho'} \backslash X_n\}$$
$$D_\mu[\![X_1 = F_1 \ldots X_n = F_n]\!]_\rho = \mu\rho' \; \rho\{\mathrm{F}[\![\phi_1]\!]_{\rho'} \backslash X_1, \ldots, [\![\phi_n]\!]_{\rho'} \backslash X_n\}$$

Table 2.5: Semantics clauses.

inductively on the structure of formulas and declarations as in Table 2.5, with $\rho \in \mathcal{E}_{V,Act}$ and $\nu$ and $\mu$ being the *maximum* respectively *minimum* fixed point operator.

For this logic the following theorem holds.

**Theorem 2.2 ([127])** *Given a formula $\phi$ it is possible to decide in exponential time in the length of $\phi$ if there exists a model of $\phi$ and it is also possible to give an example of such model.*

**Example 2.2** *By using this logic it is possible to define several security properties,* e.g., *the* Chinese Wall *policy. This policy says that, let $A$ and $B$ two sets of elements. Once one accesses to an element in $A$, he cannot access to $B$ and viceversa. Here we consider that $A$ and $B$ are sets of files and we consider the action* open. *This can be expressed by the formula $\phi = \phi_1 \vee \phi_2$ where $\phi_1$ and $\phi_2$ are the following two formulas respectively:*

$$\phi_1 = \text{LET MAX} \quad W = \quad [open_A]W \wedge [open_B]\mathbf{F}\text{IN}W$$
$$\phi_2 = \text{LET MAX} \quad V = \quad [open_B]V \wedge [open_A]\mathbf{F}\text{IN}V$$

*As a matter of fact $\phi$ is a disjunction between two different formulas $\phi_1$ and $\phi_2$ that cannot be both true at the same time. Indeed $\phi_1$ permits to open only file in $A$, on the other hand $\phi_2$ allows the access to elements in $B$.*

### Equational $\mu$-calculus

Equational $\mu$-calculus is based on fixpoint equations instead of fixpoint operators that permit to define recursively the properties of systems. A *minimal (maximal) fixpoint equation* is $Z =_\mu \phi$ ($Z =_\nu \phi$), where $\phi$ is an assertion, *i.e.*, a simple modal formula without recursion operators.

The syntax of the assertions ($\phi$) and of the lists of equations ($D$) is given by the following grammar:

$$\phi ::= Z \mid \mathbf{T} \mid \mathbf{F} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a \rangle \phi \mid [a]\phi$$

$$D ::= Z =_\nu \phi, D \mid Z =_\mu \phi, D \mid \epsilon$$

It is worthwhile noticing that the syntax of assertions is more restrictive with respect to the one for modal $\mu$-calculus. This is mainly due to our necessity to perform syntactic transformations on these assertions. This syntax permits us to keep small the size of the transformed assertions. It is assumed that variables appear only once on the left-hand sides of the equations of the list, the set of these variables will be denoted as $Def(D)$. A list of equations is closed if every variable that appears in the assertions of the list is in $Def(D)$. Let $M = \langle S, Act, \rightarrow \rangle$ be an $LTS$, $\rho$ be an environment that assigns subsets of $S$ to variables that appear in the assertions of $D$, but which are not in $Def(D)$. Then, the semantics $[\![\phi]\!]_\rho$ of an assertion $\phi$ is the same as for $\mu$-calculus assertions and the semantics $[\![D]\!]_\rho$ of a definition list is an environment which assigns subsets of $S$ to variables in $Def(D)$. As notation, we use $\sqcup$ to represent union of disjoint environments. Let $\sigma$ be in $\{\mu, \nu\}$, $\sigma U.f(U)$ represents the $\sigma$ fixpoint of the function $f$ in one variable $U$. The semantics, $[\![\mathcal{D}]\!]_\rho$ is defined by the following equations:

$$[\![\epsilon]\!]_\rho = [] \quad [\![(Z =_\sigma \phi)\mathcal{D}]\!]_\rho = [\![\mathcal{D}]\!]_{(\rho \sqcup [U'/Z])} \sqcup [U'/Z]$$

where $U' = \sigma U.[\![\phi]\!]_{(\rho \sqcup [U/Z] \sqcup \rho'(U))}$ and $\rho'(U) = [\![D]\!]_{(\rho \sqcup [U/Z])}$.

It informally says that the solution to $(Z =_\sigma \phi)D$ is the $\sigma$ fix-point solution $U'$ of $[\![\phi]\!]$ where the solution to the rest of the list of equations $D$ is used as environment. We write $M \models D \downarrow Z$ as notation for $[\![D]\!](Z)$ when the environment $\rho$ is evident from the context or $D$ is a closed list (*i.e.*, without free variables) and without propositional constants; furthermore $Z$ must be the first variable in the list $D$.

Let $\phi$ a formula. We define a positive integer, namely $ad(\phi)$ (*alternation depth*) as follows. For this it is necessary first define the notions *direct active* sub-formulas and *active* sub-formula as follows.

**Definition 2.15** *Let $\phi$ be a $\mu$-formula or a $\nu$-formula. We say that a sub-formula $\psi$ of $\phi$ is a* direct active sub-formula *of $\phi$ if $\psi \neq \phi$ and the variable $Z$ appears in $\psi$.*

It is fairly straightforward to shown that the binary relation "direct active sub-formula" is a partial order. The transitive closure of this partial order is the relation "active sub-formula".

**Definition 2.16** *A formula $\psi$ is an* active sub-formula *of $\phi$, if there exists a sequence (or a chain) of sub-formulas $\varphi_1, \varphi_2, \ldots, \varphi_k$ such that $\phi = \varphi_1$, $\psi = \varphi_k$ and for each $i$, $1 \leq i < k$ $\varphi_{i+1}$ is a direct active sub-formula of $\varphi_i$.*

**Definition 2.17** *It is possible to define $ad(\phi)$ as follows:*

- *For a $\mu$-formula $\phi$, $ad(\phi) = 0$ if $\phi$ has no active $\nu$-sub-formulas in it, otherwise $ad(\phi) = 1 + max\{ad(\psi) : \psi$ is an active $\nu$-sub-formula of $\phi\}$.*
- *For a $\nu$-formula $\phi$, $ad(\phi) = 0$ if $\phi$ has no active $\mu$-sub-formulas in it, otherwise $ad(\phi) = 1 + max\{ad(\psi) : \psi$ is an active $\mu$-sub-formula of $\phi\}$.*

- *For any formula $\phi$ define $ad(\phi) = max\{ad(\psi) : \psi$ is a $\mu$-sub-formula or a $\nu$-sub-formula of $\phi\}$.*

Moreover we said that $\phi$ is an *alternation-free* formula if $ad(\phi) = 0$.

**Examples and facts** A lot of properties can be defined by using equational $\mu$-calculus. It is possible to translate from a modal $\mu$-calculus formula to an equational one and vice-vera. For that reason, we can express the same properties. In particular it is useful to express several security properties. In order to underline that equational $\mu$-calculus formulas are more concise than in modal $\mu$-calculus, we shows how in equational $\mu$-calculus could be expressed the safety property that expresses the possibility to open a new file only if the previous one is closed:

$$Z_1 =_\nu [\texttt{open}]([\texttt{close}]Z_1 \wedge [\texttt{open}]\mathbf{F})$$

The same consideration can be done for the specification of a *liveness property*. For instance, the property "a state satisfying $\phi$ can be reached" is expressed by $Z =_\mu \langle\_\rangle Z \vee \phi$[3].

Also for this calculus we have a satisfiability result similar to Theorem 2.2.

### 2.2.3 Temporal Logic

The term *temporal logic* is used to describe any system of rules and symbolisms for representing, and reasoning about, propositions qualified in terms of time.

For instance, it seems reasonable to say that possibly it will rain tomorrow, and possibly it won't; on the other hand, if it rained yesterday, if it really already did so, then it cannot be quite correct to say "It may not have rained yesterday". It seems that the past is "fixed" or necessary, in a way that the future is not. This aspect is sometimes referred to as accidental necessity.

A standard method for formalizing time is to use two pairs of operators, one for the past and one for the future. For the past, let "*It has always been the case that ...*" be equivalent to the box of modal logic, and let "*It was once the case that ...*" be equivalent to the diamond of modal logic. For the future, let "*It will always be the case that ...*" be equivalent to the box of the modal logic, and let "*it will eventually be the case that ...*" be equivalent to the diamond of modal logic.

Temporal logics may differ about the underlying nature of time which is assumed: if time has only a single possible future moment we call them *linear* time logics; otherwise, if there may be many possible future moments, we call them *branching* time logics. Temporal operators of the logic typically reflect the underlying nature of time, so *linear* time temporal logics have operators for expressing properties about a single time line, while *branching* time logics have also operators that permit to quantify along possible time lines

[3]In writing properties, here and in the rest of the paper, we use the shortcut notations $[\_]$ means $[Act]$ and, equivalently, $\langle\_\rangle$ means $\langle Act\rangle$.

(also referred to as computation tree) that may start from a time point. Modal logic that we have presented in the previous section are branching time logics.

Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems.

In his landmark work (see [112]) Pnueli proposed *temporal logics* as a well-suited formalism to reason about *reactive* system behavior, *i.e.*, systems that keep an *ongoing* interaction with the environment by receiving and emitting stimuli. His intuition has been followed by many researchers. Since non termination is, usually, one of the main characteristics of concurrent systems, it is reasonable to search for a formalism whose operators can express properties about possibly non terminating executions of systems. Temporal logic modalities permit to reason about executions (computations) of systems.

### Linear Temporal Logics

*Linear Temporal Logic*, $LTL$ for short, provides an important framework for formally specifying systems and reasoning about them. Usually, specification and verification are done in *pure future* temporal logics, *i.e.*, logics where the modalities only refer to the future of the current time. It is well-known that temporal logics combining past and future modalities make some specifications easier to write and more natural (see [80]). All the main logics with past-time admit translation to their pure-future fragment (see [55, 56]).

The $LTL$ is built up from a set of proposition variables $AP = \{p1, p2, ...\}$, the usual logic connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and the following temporal modal operators, $\mathtt{N}$, $\mathtt{G}$, $\mathtt{F}$ and $\mathtt{U}$. So the grammar is the following:

$$\phi ::= AP \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathtt{N}\phi \mid \mathtt{F}\phi \mid \mathtt{G}\phi \mid \phi_1 \mathtt{U} \phi_2$$

where $\mathtt{U}$ reads *until* and $\mathtt{N}$ is *next*. Moreover, we have $\mathtt{F}\phi$ (*sometimes* $\phi$), $\mathtt{G}\phi$ (*always* $\phi$).

It is easy to note that the $LTL$ grammar can be reduced using only $N$ and $U$ as temporal operators. Let $n = length(u) = |u|$ be the length of $u$. Let $u = a_0 a_1 \ldots$ be a path, we denote by $u^{[0]}$ the initial state of the sequence, by $u^{[i]}$ the $i^{th}$ state and by $u^{[0\ldots i]}$ its prefix which length is $i$.

We first recall the notion of *equivalence* between formulas. We write $\phi_1 \equiv \phi_2$ when $\phi_1$ and $\phi_2$ are *equivalent*, *i.e.*, when for all $u$ and $i$, we have $u^{[i]} \models \phi_1$ if and only if $u^{[i]} \models \phi_2$. A less discriminating equivalence is *initial equivalence*, denoted $\equiv_i$, and defined by: $\phi_1 \equiv_i \phi_2$ if and only if for all $u$, $u^{[0]} \models \phi_1$ if and only if $u^{[0]} \models \phi_2$.

"Global" equivalence ($\equiv$) is the natural notion of equivalence, and it is substitutive.

The following equivalences hold:

$$\begin{array}{lcl}
\mathbf{F} & \equiv & p \wedge \neg p \\
\mathbf{T} & \equiv & \neg\mathbf{F} \\
\phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \Rightarrow \phi_2 & \equiv & \neg\phi_1 \vee \phi_2 \\
\phi_1 \Leftrightarrow \phi_2 & \equiv & (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \\
\mathtt{F}\phi & \equiv & \mathbf{T}\mathtt{U}\phi \\
\mathtt{G}\phi & \equiv & \neg\mathtt{F}\neg\phi
\end{array}$$

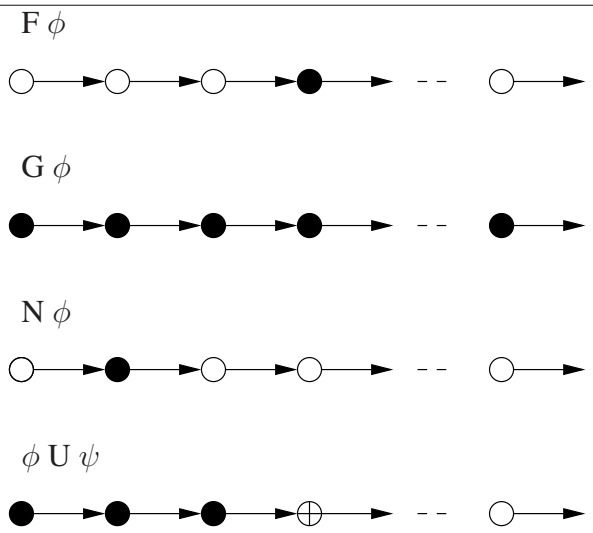



Figure 2.2: Intuition for linear-time operators. In the figure, the black states satisfy formula $\phi$ and the crossed state satisfy $\psi$.

An $LTL$ formula can be evaluated over a sequence of truth evaluations and a position on that path. Let $u$ be a path, $i$ a non-negative integer and $\phi$ a $LTL$ formula. Notation $u^{[i]} \models \phi$ means that "the formula $\phi$ holds in the position $i$ of $u$". The semantics for the modal operators it is interpreted on a Kripke structure and it is given as follows.

$$
\begin{array}{lll}
u^{[i]} \models p & \text{iff} & p \in \delta(u,0) \\
u^{[i]} \models \phi_1 \wedge \phi_2 & \text{iff} & u^{[i]} \models \phi_1 \quad \text{and} \quad u^{[i]} \models \phi_2 \\
u^{[i]} \models \neg\phi & \text{iff} & \text{not} \quad u^{[i]} \models \phi \\
u^{[i]} \models \mathtt{N}\phi & \text{iff} & u^{[i+1]} \models \phi \\
u^{[i]} \models \psi \mathtt{U} \phi & \text{iff} & \exists k \geq 0 \text{ s.t. } u^{[k]} \models \phi \quad \text{and} \quad \forall 0 \leq j < k \quad u^{[j]} \models \psi
\end{array}
$$

A graphical explanation of $LTL$ operators is shown in Figure 2.2.

We can then define the other temporal operators as derived ones. Among the derived temporal operators there is also $\psi \mathtt{B} \phi$ ("$\psi$ before $\phi$"), which expresses the fact that during a time line the formula $\psi$ is true before the formula $\phi$, this abbreviates the formula $\neg(\neg\psi \mathtt{U} \phi)$. Another one is $\phi_1 \mathtt{R} \phi_2$, reads *release*, that can be derived from `U` as $\neg(\neg\phi_1 \mathtt{U} \neg\phi_2)$.

Below we give some simple equivalences among $LTL$ formulas:

$$
\begin{array}{ll}
(1) & \models \mathtt{G} \quad \mathtt{G}\phi \Leftrightarrow \mathtt{G}\phi \\
(2) & \models \mathtt{F} \quad \mathtt{F}\phi \Leftrightarrow \mathtt{F}\phi \\
(3) & \models \phi \Rightarrow \mathtt{F}\phi \\
(4) & \models \mathtt{G}\phi \Rightarrow \phi \\
(5) & \models \mathtt{F}\phi \Leftrightarrow \phi \vee \mathtt{NF}\phi \\
(6) & \models \mathtt{G}\phi \Leftrightarrow \phi \wedge \mathtt{NG}\phi \\
(7) & \models \psi \mathtt{U} \phi \Leftrightarrow \phi \vee (\psi \wedge \mathtt{N}(\psi \mathtt{U} \phi))
\end{array}
$$

The last three equivalences are called fixpoint characterization of temporal operators in terms of "next" and "until" operators. For example the equivalence (5) can be explained by noticing that if $\mathtt{F}\phi$ holds in a time line then $\phi$ holds at the first instant of time or $\phi$ holds at a future moment and hence $\mathtt{F}\phi$ holds in the time line that starts from the next time. On the other hand, if $\phi$ holds at the first instant of the time line or $\mathtt{F}\phi$ holds in the time line that starts from the next instant then obviously $\mathtt{F}\phi$ holds in the time line.

**Linear temporal logic with Past ($PLTL$)** In addition to the temporal operator $N$ and $U$ of $LTL$, that express properties in the future, we consider also operators that describe the behavior of the system in the past. Hence, the grammar of $PLTL$ is the grammar of $LTL$ with the operators $\mathtt{S}$ (*Since*), $\mathtt{N}^{-1}$ (*Previously*). From these two operators can be derived other two operators: $\mathtt{F}^{-1}$ and $\mathtt{G}^{-1}$ as follows:

$$\mathtt{F}^{-1}\phi \equiv \mathbf{T}\mathtt{S}\phi$$
$$\mathtt{G}^{-1}\phi \equiv \neg\mathtt{F}^{-1}\neg\phi$$

Now we give the semantics of $PLTL$ by using Kripke structure.

The truth relation ($\models$) is defined inductively in the structure of the formula $\phi$.

$$u^{[i]} \models \mathtt{N}^{-1}\phi \quad \text{iff} \quad u^{[i-1]} \models \phi$$
$$u^{[i]} \models \psi\mathtt{S}\phi \quad \text{iff} \quad \exists j \leq i \quad u^{[j]} \models \phi \text{ and } \forall j < k \leq i \quad u^{[k]} \models \psi$$

We say that a $PLTL$ formula $\phi$ is *satisfiable* if and only if there exists a linear-time structure $(u, \delta)$ such that $u^{[i]} \models \phi$. We say that any such structure is a *model* for $\phi$. We say $\phi$ is valid, and write $\models \phi$, if and only if for all linear structures $(u, \delta)$ such that $u^{[i]} \models \phi$.

Gabbay's theorem, stating that "any $PLTL$ formula can be translated into an equivalent $LTL$ formula", refers to initial equivalence: saying that $\phi_1\mathtt{U}\phi_2$ and $\mathtt{F}(\phi_2 \wedge \mathtt{G}^{-1}(\phi_1 \vee \phi_2))$ are equivalent is only correct with initial equivalence in mind.

## 2.3 Process algebra

*Process algebras* (see [65, 67]) are approaches to formally model concurrent systems. They provide a method for the high-level description of interactions, communications, and synchronization between independent entities.

Thus, rather than actual programming languages, *process algebras* are specification formalisms for systems that have to cooperate and communicate to perform complex tasks and computations in different settings and in different contexts. The universe of interest is modeled by assuming the notion of *processes* that autonomously and concurrently can proceed in their computation but which have also the possibility to communicate and synchronize among themselves. *Process algebra* formalisms are built from the basic operations of this framework. The processes can perform actions which may represent computation steps.

A process calculus of our interest is the *Calculus of Communicating Systems* [103], $CCS$ for short, developed by Robin Milner and presented in Section 2.3.2. Its actions model invisible communications between two participants. The notion of communication considered is a synchronous one, *i.e.*, both processes must agree on performing the communication at the same time.

### 2.3.1 Operational semantics

The importance of giving precise semantics to programming and specification languages was recognized since the sixties with the development of the first high-level programming languages.

In this section we introduce a formal method for giving (operational) semantics to programs, namely *Structural Operational Semantics* ($SOS$, for short) proposed by Plotkin (see [111]). The operational semantics explicitly describes how programs compute in a stepwise fashion, and the possible state-transformations they can perform. Moreover, this method has a logical flavor and permits to compositionally reason about the behavior of programs.

This method is based on the notion of labeled transition systems, introduced in Section 2.2.1. The states of the transition system are the elements of some formal language. The key point is the transition between states, *i.e.*, $s \xrightarrow{a} s'$, which expresses the fact that the system, initially in state $s$, performs the action labelled $a$, and it reachs the state $s'$. In general, transitions can be inferred through a set of (conditional) rules, based on the syntax of the language. As an example, the transition $s \xrightarrow{a} s'$ can be derived by inspecting the transitional behavior of subcomponents of $s$. The general structure for a $SOS$ rule is as follows:

$$\frac{premises}{conclusions}$$

where *premises* and *conclusions* are properties expressed on transitions of open terms of the language. The semantics of terms can be inferred by the semantics of the subterms. Many important facts about programming languages, whose operational semantics is given in terms of $SOS$ rules, can be deduced simply by inspecting the format of these rules. For example, it is possible to ensure that an equivalence relation, section 2.3.3, among closed terms is indeed a congruence with respect to the operators of the language, [25]. Moreover, a lot of work has been carried out in order to automatically derive complete equational theories from $SOS$ specifications for (concurrent) programming languages, see [1].

### 2.3.2 A process algebra: $CCS$

The main notion of $CCS$ is the communication between processes, that is a synchronous one. Both the processes must agree on performing the communication at the same time, and communication is modeled by a simultaneous performing of complementary actions

(*e.g.*, send-receive actions). This event is represented by a synchronization action (or internal action) $\tau$.

The main operator is the *parallel composition* between processes, namely $P \| Q$. The intuition is that the parallel composition of two processes performs an action whenever anyone of the two processes performs an action. Moreover, processes can communicate. The $CCS$ language assumes a set $Act = \mathcal{L} \cup \bar{\mathcal{L}} \cup \{\tau\}$ of *communication actions* built from a set $\mathcal{L}$ of names and a set $\bar{\mathcal{L}}$ of co-names. Putting a line, called complementation, over a name means that the corresponding action can synchronize with its complemented action. Complementation follows the rule that $\bar{\bar{a}} = a$, for any communication action $a \in Act$. The special symbol, $\tau$, is used to model any (unobservable) *internal action*. We let $a, b, \ldots$ range over $Act$.

The following grammar specifies the syntax of the language defining all $CCS$ processes:

$$P, Q ::= \mathbf{0} \mid a.P \mid P + Q \mid P \| Q \mid P \backslash L \mid P[f] \mid A$$

where $L \subseteq Act$ and the relabeling function $f : Act \mapsto Act$ must be such that $f(\tau) = \tau$.

The operational semantics of $CCS$ terms (see [103]) is described by a labeled transition system $(\mathcal{E}, Act, \rightarrow)$, where $\mathcal{E}$ is the set of all $CCS$ terms and $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation defined by structural induction as the least relation generated by the set of the structural operational semantics rules of Table 2.6. The transition relation $\rightarrow$ defines the usual concept of derivation in one step. As a matter of fact $P \xrightarrow{a} P'$ means that process $P$ evolves in one step into process $P'$ by executing action $a \in Act$. The transitive and reflexive closure of $\bigcup_{a \in Act} \xrightarrow{a}$ is written $\rightarrow^*$.

Informally, the meaning of $CCS$ operators is the following:

**0:** is the process that does nothing.

**Prefix:** a (closed) term $a.P$ represents a process that performs an action $a$ and then behaves as $P$.

**Choice:** the term $P + Q$ represents the non-deterministic choice between the processes $P$ and $Q$. Choosing the action of one of the two components means dropping the other.

**Parallel composition:** the term $P \| Q$ represents the parallel composition of $P$ and $Q$. It can perform an action if one of the two processes can perform that action, and this does not prevent the capabilities of the other process. The third rule of parallel composition is characteristic of this calculus, it expresses that the communication between processes happens whenever both can perform complementary actions. The resulting process is given by the parallel composition of successors of each component, respectively.

**Restriction:** the process $P \backslash L$ behaves like $P$ but the actions in $L \cup \bar{L}$ are forbidden. To force a synchronization on an action between parallel processes, we have to set restriction operator in conjunction with parallel one.

Prefixing:

$$\frac{}{a.P \xrightarrow{a} P}$$

Choice:

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

Parallel:

$$\frac{P \xrightarrow{a} P'}{P \| Q \xrightarrow{a} P' \| Q} \qquad \frac{Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P \| Q'} \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'}$$

Restriction:

$$\frac{P \xrightarrow{a} P'}{P \backslash L \xrightarrow{a} P' \backslash L}$$

Relabeling:

$$\frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{f(a)} P'[f]}$$

Constant:

$$\frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}$$

Table 2.6: *SOS* system for $CCS$.

**Relabeling:** the process $P[f]$ behaves like $P$, but its actions are renamed through relabeling function $f$.

**Constant:** $A$ defines a process and it is assumed that each constant $A$ has a defining equation of the form $A \doteq P$.

Given a $CCS$ process $P$, $Der(P) = \{P' | P \rightarrow^* P'\}$ is the set of its derivatives. A $CCS$ process $P$ is said *finite state* if $Der(P)$ is finite. $Sort(P)$ is the set of names of actions that syntactically appear in the process $P$.

### A timed variant

Several languages have been developed in the literature to describe systems in a timed setting, see, *e.g.*, [6, 61, 116]. Here, we present a timed variant of $CCS$, called $timedCCS$ (see [116]). This approach considers that time is discrete, actions are durationless and there is one special $tick$ action to represent the elapsing of time. These are features of the so called *fictitious clock* approach, *e.g.*, [35, 66, 129]. A global clock is supposed to be updated whenever all processes of the system agree on it, by globally synchronizing

Prefixing:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Choice:

$$\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \quad \frac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'} \quad \frac{P \xrightarrow{tick} P' \quad Q \xrightarrow{tick} Q'}{P+Q \xrightarrow{tick} P'+Q'}$$

Parallel:

$$\frac{P \xrightarrow{a} P'}{P_1 \| Q \xrightarrow{a} P' \| Q} \qquad \frac{Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P \| Q'} \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'}$$

$$\frac{P \xrightarrow{tick} P' \quad Q \xrightarrow{tick} Q'}{P \| Q \xrightarrow{tick} P' \| Q'}$$

Restriction:

$$\frac{P \xrightarrow{\alpha} P'}{P \backslash L \xrightarrow{\alpha} P' \backslash L} (\alpha \notin L \cup \overline{L})$$

Idling:

$$\frac{P \not\xrightarrow{tick} \quad P \not\xrightarrow{\tau}}{i(P) \xrightarrow{tick} i(P)} \qquad \frac{P \xrightarrow{tick} P'}{i(P) \xrightarrow{tick} i(P')} \qquad \frac{P \xrightarrow{\alpha} P'}{i(P) \xrightarrow{\alpha} P'}$$

Table 2.7: Operational semantics for $timedCCS$ (see [116]).

on action $tick$. Hence, between two global synchronizations on action $tick$ all processes proceed asynchronously by performing durationless actions.

The set of $timedCCS$ processes is denoted with $\mathcal{E}_t$, ranged over by $E_t, F_t, P_t, Q_t \ldots$[4]. Let $Act_t = \mathcal{L} \cup \bar{\mathcal{L}} \cup \{\tau\} \cup \{tick\}$ be the set of actions, ranged over by $\alpha, \beta, \ldots$ and let $\mathcal{L} \cup \{\tau\}$ be the actions ranged over by $a, b, c, \ldots$.

The semantics is given in Table 2.7. Here, we omit to describe rules already presented for standard $CCS$. $tick.P$ lets one time unit pass. $P+Q$ represents the nondeterministic choice between the two processes $P$ and $Q$; time passes when both $P$ and $Q$ are able to perform a $tick$ action and, in such a case, by performing $tick$, a configuration where both the derivatives of the summands can still be chosen is reached. The *parallel* operator $P \| Q$ is the core operator for time. As a matter of fact both components must agree on performing a *tick* action. $i(P)$ (*idling*) allows process $P$ to wait indefinitely. At every instant of time, if process $P$ performs an action $\alpha$ then the whole system proceeds in this state, while dropping the idling operator.

[4]In the rest of the thesis we will write $P$ instead of $P_t$ whenever it is clear from the context if we are working in a timed setting or not.

**Example and facts** By using $timedCCS$ we are able to model temporal aspects of systems. For instance, we can easily model *timeout* constructs. Let us assume $n_1 \leq n_2$. We define the following process:

$$\text{TIME\_OUT}(n_1, n_2, P, Q) = tick^{n_1}.i(P) + tick^{n_2}.\tau.Q$$

where $P$ and $Q$ are two processes. $\text{TIME\_OUT}(n_1, n_2, P, Q)$ first performs a sequence of $n_1$ $tick$ actions; then, the system may perform $n_2 - n_1$ $tick$ actions, unless $P$ resolves the choice by performing an action; instead if $P$ does not do anything, after $n_2$ time units, via the execution of a $\tau$ action, the process is forced to act as $Q$.

### 2.3.3 Behavioral equivalence

In the literature, many different equivalence theories have been proposed, due to the huge number of different settings that arise in the analysis of concurrent systems.

In general, it is interesting to study when two processes (terms) can be considered equivalent, by abstracting from irrelevant aspects. What a relevant aspect is, mainly depends on the way a process is used, as well as on the identification of the properties that it should satisfy. Furthermore, certain equivalence notions may preserve some properties, while others may not. Also, when considering Labeled Transition Systems, it is important to consider the *behavioral* capacity of the system to react with the outside world, rather than its internal state.

Here, we briefly introduce some well known equivalences among processes (for a deeper discussion one can see, *e.g.*, [26, 39, 40, 130]).

Within a computer security field, most of the security properties are based on the simple notion of *traces*: two processes are equivalent if they exactly show the same execution sequences (called *traces*). In order to formally define traces, we define *trace pre-order* ($\leq_{trace}$) and *trace equivalence* ($\approx_{trace}$) as follows.

**Definition 2.18** *For any $P \in \mathcal{E}$ the set $T(P)$ of* traces associated with $P$ *is* $T(P) = \{\gamma \in Act^* \mid \exists P' : P \xrightarrow{\gamma} P'\}$*, where $Act^*$ is the set of sequences of actions and, let $\gamma$ be the sequence $a_1 a_2 \ldots a_n$, $\xrightarrow{\gamma} = \xrightarrow{a_1}\xrightarrow{a_2} \ldots \xrightarrow{a_n}$. $Q$ can execute all traces of $P$ (notation $P \leq_{trace} Q$) if and only if $T(P) \subseteq T(Q)$. $P$ and $Q$ are* trace equivalent *(notation $P \approx_{trace} Q$) if and only if $P \leq_{trace} Q$ and $Q \leq_{trace} P$,* i.e.*, if and only if $T(P) = T(Q)$.*

This equivalence, even though rather intuitive, is not completely satisfactory from several points of view. Concurrent systems may present deadlocks, *i.e.*, the system cannot proceed and cannot perform its task. The above equivalence does not take into account deadlocks, *e.g.*, the following terms are considered equivalent under a trace equivalence notion: $a.\mathbf{0} + a.b.\mathbf{0}$ and $a.b.\mathbf{0}$. Actually, while the second process always reaches the deadlock state upon sequentially performing $a$ and $b$, the first process could reach a deadlock configuration not only performing actions $a$, $b$, but it can also perform only $a$, given the presence of the non-deterministic choice.

Another negative aspect about trace equivalence is that it does not take into account the branching structure of the processes. For this reason, another notion of pre-order and equivalence is introduced: the *simulation*. Let us consider the following example.

**Example 2.3** *Consider two vendor machines $P$ and $Q$ which behaviors can be represented by their $LTSs$ (see Figure 2.3).*

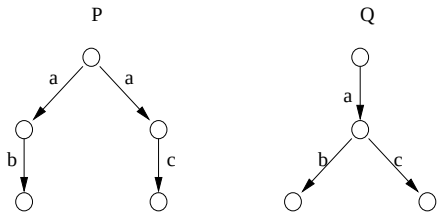



Figure 2.3: Example of two similar processes.

*The first process "chooses" to perform a $b$ or $c$ action at the beginning of its computation, while the latter after performing an $a$ action; if this action can influence the choice of the following behavior of the process then it is reasonable to consider that the second process has a more decisional power.*

To underline the way in which the processes in Figure 2.3 differ, we introduce the notion of *simulation*, according to which $Q$ can simulate $P$, but the contrary does not hold. Informally, saying that "$Q$ simulates $P$" means that $Q$'s behavior pattern is at least as rich as that of $P$.

More formally, we can define the notion of *strong bisimulation* by following Park's definition [107].

**Definition 2.19** *Let $(\mathcal{E}, Act, \rightarrow)$ be an $LTS$ of concurrent processes over the set of actions $Act$, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* strong simulation*, denoted by $\prec$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

$$\text{if } P \xrightarrow{a} P' \text{ then } \exists\, Q' \text{ such that } Q \xrightarrow{a} Q' \text{ and } (P', Q') \in \mathcal{R}$$

Recalling that the converse $\mathcal{R}^{-1}$ of any binary relation $\mathcal{R}$ is the set of pairs $(Q, P)$ such that $(P, Q)$ are in $\mathcal{R}$, we give the following definition.

**Definition 2.20** *A* strong bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations. We represent with $\sim$ the union of all the strong bisimulations.*

Two processes $P$ and $Q$ are strong bisimilar if there exists a strong bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. The maximal strong bisimulation is $\sim$ which is the union of every strong bisimulation. It is easy to check that this relation is still a strong bisimulation and moreover is reflexive, symmetric and transitive.

Strong bisimulation is the finest equivalence that is commonly accepted and enjoys several good properties. First of all, this equivalence is also a congruence with respect to all $CCS$ operators.

Another important aspect of bisimulation is that it can be logically characterized.

**Proposition 2.3 ([102])** *If $P$ and $Q$ are finitely branching processes then*

$$P \sim Q \text{ if and only if } \forall \phi \in HML\ (P \models \phi \Leftrightarrow Q \models \phi).$$

This strong connection between modal logics and models of our concurrent languages is one of the major advantages in considering interleaving semantics for concurrency. This connection plays a central role in this thesis. As a matter of fact, by investigating on the behavioral relations existing between given processes, we are able to synthesize secure system by generating controller process by applying satisfiability procedure for temporal logic.

### Observational equivalence or weak bisimulation

Up to now, we do not have assumed a distinguished role for the $\tau$ action. This action has been used to model an internal communication within the system, or an internal computation step, not visible to the outside world. We may want to abstract from those actions when comparing two systems. Within a step-wise development strategy, this could be appealing, because we would be able to substitute more complex specifications with simpler ones, without however affecting the overall visible behavior of the system. For example, we can imagine to substitute a process with two others that perform the same visible task, but omitting some internal communication. The point is that we cannot simply abstract from the internal actions, since they also can affect the visible behavior of a system. Look at the following figure:

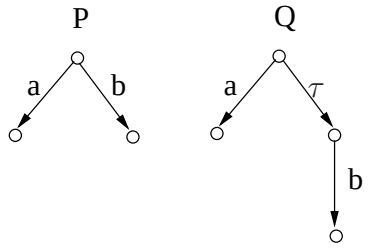



Figure 2.4: Example of two not observationally bisimilar processes.

The processes $P$ and $Q$ cannot be consider equivalent, since the second performs an internal action by reaching a state where an action $a$ is no longer possible. Thus, the non visible behavior of the system, represented by the $\tau$ action, can modify its visible behavior. To

compare this kind of processes, Milner, in [103], proposed the notion of *observational equivalence*, or *weak bisimulation*.

Let us consider $a \neq \tau$, $\hat{a} = a$, and $\hat{\tau} = \epsilon$. Then, we use the notation $P \stackrel{\tau}{\Longrightarrow} P'$ in order to denote that $P$ and $P'$ belongs to the reflexive and transitive closure of $\tau$. The same holds for notation $P \stackrel{\epsilon}{\Longrightarrow} P'$. Also, $P \stackrel{\hat{a}}{\Rightarrow} P'$ if $P \stackrel{\epsilon}{\Rightarrow} P_\epsilon \stackrel{\hat{a}}{\rightarrow} P'_\epsilon \stackrel{\epsilon}{\Rightarrow} P'$ where $P_\epsilon$ and $P'_\epsilon$ denote intermediate states[5].

We can give the following definition:

**Definition 2.21** *Let $(\mathcal{E}, Act, \rightarrow)$ be an $LTS$ of concurrent processes over the set of actions $Act$, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* weak simulation*, denoted by $\preceq$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

$$\textit{if } P \stackrel{a}{\longrightarrow} P' \textit{ then } \exists\, Q' \textit{ such that } Q \stackrel{a}{\Longrightarrow} Q' \textit{ and } (P', Q') \in \mathcal{R}$$

Recalling that the converse $\mathcal{R}^{-1}$ of any binary relation $\mathcal{R}$ is the set of pairs $(Q, P)$ such that $(P, Q)$ are in $\mathcal{R}$, we give the following definition.

**Definition 2.22** *A* weak bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations,* i.e.*, if for each $(P, Q) \in \mathcal{R}$ and for each $a \in Act$:*

*if $P \stackrel{a}{\longrightarrow} P'$ then there exists $Q' : Q \stackrel{a}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$.*

*if $Q \stackrel{a}{\longrightarrow} Q'$ then there exists $P' : P \stackrel{a}{\Longrightarrow} P'$ and $(P', Q') \in \mathcal{R}$.*

Two processes $P$ and $Q$ are weakly bisimilar if there exists a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. The maximal weak bisimulation is $\approx$ which is the union of every weak bisimulation. It is easy to check that this relation is still a weak bisimulation and moreover is reflexive, symmetric and transitive. Weak bisimulation is a congruence with respect to all $CCS$ operators, except summation (+).

An important result proved by Milner is the following.

**Proposition 2.4 ([103])** *Every strong simulation is also a weak one.*

**Example 2.4** *Let us consider the processes $E$, $F$ and $P$ of Fig. 2.5. $F$ and $P$ are weakly bisimilar, while $E$ and $F$ ($P$) are not.*

Bisimulation is a very interesting equivalence. It is decidable in polynomial time for finite-state processes, [70]. Moreover, proving that two processes $P$ and $Q$ are bisimilar can be done by quite elegant proof techniques. Actually, it is sufficient to provide a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.

[5] We can use the short notation $P \stackrel{\epsilon}{\Longrightarrow} \stackrel{\hat{a}}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} P'$ when the intermediate states are not relevant.

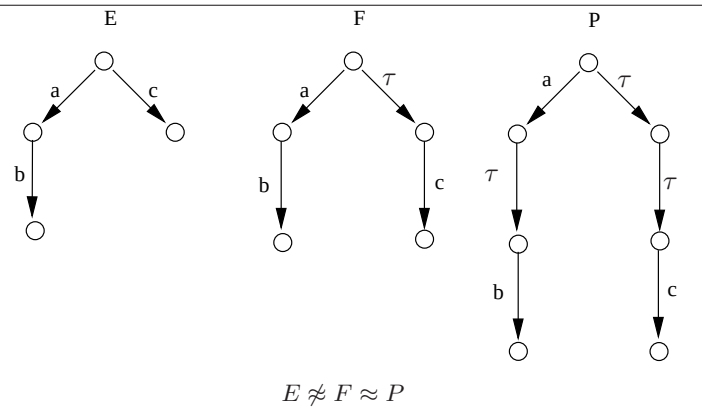



Figure 2.5: Example of observational equivalence between different processes.

**Observational equivalences in a timed setting**

The definitions given in the previous section can be extended to deal with processes in a timed setting. We follow the approach of [116].

**Definition 2.23** *Let $(\mathcal{E}, Act, \rightarrow)$ be a $LTS$ of concurrent processes and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* timed strong simulation $\prec_t$ *over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$, we have:*

- *if $P \xrightarrow{a} P'$ then there exists $Q'$ such that $Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$,*
- *if $P \xrightarrow{tick} P'$ then there exists $Q'$ such that $Q \xrightarrow{tick} Q'$ and $(P', Q') \in \mathcal{R}$.*

*A* timed strong bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are timed strong simulations. We represent with $\sim_t$ the union of all the timed strong bisimulations.*

For our purposes, it is also useful to consider those processes allowing time to pass, the so-called *weakly time alive* processes.

**Definition 2.24** *A process $P$ is* directly weakly time alive *if and only if $P \stackrel{tick}{\Longrightarrow}$* [6]*, while it is* weakly time alive *if and only if for all $P' \in Der(P)$, $P'$ is directly weakly time alive.*

Since $P \xrightarrow{\alpha} P'$ implies $Der(P') \subseteq Der(P)$, it directly follows that if $P$ is weakly time alive, then any derived $P'$ of $P$ is weakly time alive as well. Moreover, it is worthwhile noticing that the above property is preserved by the parallel composition.

We define the *timed weak bisimulation* relation as follows.

**Definition 2.25** *Let $(\mathcal{E}, Act, \rightarrow)$ be a $LTS$ of concurrent processes and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* timed weak simulation $\preceq_t$ *over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

[6] We are no interested to the final state of the transition.

- *if* $P \xrightarrow{a} P'$ *then there exists* $Q'$ *such that* $Q \stackrel{a}{\Longrightarrow} Q'$ *and* $(P', Q') \in \mathcal{R}$,
- *if* $P \xrightarrow{tick} P'$ *then there exists* $Q'$ *such that* $Q \stackrel{tick}{\Longrightarrow} Q'$ *and* $(P', Q') \in \mathcal{R}$.

*A* timed weak bisimulation *is a relation* $\mathcal{R}$ *such that both* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are timed weak simulations. We represent with* $\approx_t$ *the union of all the timed weak bisimulations.*

Also, a proposition similar to Proposition 2.4 holds for the timed setting.

### Characteristic formulas

Finite-state processes can be characterized by equational $\mu$-calculus formulas with respect to strong and weak bisimulation. This characterization can be derived from the greatest fixpoint characterization of the bisimulation relation.

A *characteristic formula* is a formula in equational $\mu$-calculus that completely characterizes the behavior of a state-transition graph or of a state in a graph modulo a chosen notion of behavioral relation.

Here, we recall the definition of the characteristic formula of a finite-state process, by following the approach studied in [105].

**Definition 2.26** *Given a finite-state process* $P$*, its* characteristic formula *with respect to strong bisimulation is given by the closed list* $D_P \downarrow Z_P$ *where for every* $P' \in Der(P)$*,* $a \in Act$*:*

$$Z_{P'} =_\nu \Big( \bigwedge_{a \in Act; P'' : P' \xrightarrow{a} P''} \langle a \rangle Z_{P''} \Big) \wedge \Big( \bigwedge_{a \in Act} \big( [a] \big( \bigvee_{P'' : P' \xrightarrow{a} P''} Z_{P''} \big) \big) \Big)$$

Strong bisimulation requires that every step of a process is matched by a corresponding step of a bisimilar process. Considering weak bisimulation, this requirements is relaxed, since internal actions of a process can be matched by zero or more internal steps of the other process.

Let $\langle\langle a \rangle\rangle$ be a weak version of the modality $\langle a \rangle$, introduced as abbreviation and defined as follows (see [105]):

$$\langle\langle \epsilon \rangle\rangle \phi \stackrel{def}{=} \mu Z. \phi \vee \langle \tau \rangle Z \qquad \langle\langle a \rangle\rangle \phi \stackrel{def}{=} \langle\langle \epsilon \rangle\rangle \langle a \rangle \langle\langle \epsilon \rangle\rangle \phi$$

Now we are able to give the following definition.

**Definition 2.27** *Given a finite-state process* $P$*, its characteristic formula with respect to weak bisimulation is given by the closed list* $D_P \downarrow Z_P$ *where for every* $P' \in Der(P)$*,* $a \in Act$*:*

$$Z_{P'} =_\nu \Big( \bigwedge_{a \in Act; P'' : P' \xrightarrow{a} P''} \langle\langle \hat{a} \rangle\rangle Z_{P''} \Big) \wedge \Big( \bigwedge_{a \in Act} \big( [a] \big( \bigvee_{P'' : P' \stackrel{\hat{a}}{\Rightarrow} P''} Z_{P''} \big) \big) \Big)$$

The following lemma characterizes the power of these formulas.

**Lemma 2.2 ([116])** *Let $P_1$ and $P_2$ be two different finite-state processes. If $\phi_{P_2}$ is characteristic for $P_2$ then:*

1. *If $P_1 \approx P_2$ then $P_1 \models \phi_{P_2}$;*
2. *If $P_1 \models \phi_{P_2}$ and $P_1$ is finite-state then $P_1 \approx P_2$.*

Following a similar reasoning to the one in [105] for defining characteristic formulas with respect to bisimulation, it is possible to define a characteristic formula for a given process also with respect to strong and weak simulation. Let us start from strong simulation. Let $\mathcal{R}$ be the strong simulation relation, let $Z_{P_i}$ and $Z_{Q_i}$ be the variables associated to each derivatives of $P$ and $Q$, let $D_P \downarrow Z_P$ the list of equation, one for each derivative of $P$ and let $M_P(D_P)(Z_{Q'}) = \{P' \in Der(P) | (P', Q') \in \mathcal{R}\}$. Then, two processes $P$ and $Q$ are strongly similar, *i.e.*,

$$
\begin{array}{l}
(P, Q) \in \mathcal{R} \\
\text{iff} \\
\forall a \quad P \xrightarrow{a} P' \quad \exists Q' \quad Q \xrightarrow{a} Q' \wedge (P', Q') \in \mathcal{R} \\
\text{iff [Definition of } M_P(Z_{Q'})\text{]} \\
\forall a \quad P \xrightarrow{a} P' \quad \exists Q' \quad Q \xrightarrow{a} Q' \wedge P' \in M_P(Z_{Q'}) \\
\text{iff [Definition of } \bigvee\text{]} \\
\forall a \quad P \xrightarrow{a} P' \quad P' \in M_P(\bigvee\{Z_{Q'} | Q \xrightarrow{a} Q'\}) \\
\text{iff [Definition of } [a]\text{]} \\
\forall a \quad P \in M_P([a] \bigvee_{a \in Act}\{Z_{Q'} | Q \xrightarrow{a} Q'\}) \\
\text{iff [Definition of } \bigwedge\text{]} \\
P \in M_P(\bigwedge_{a \in Act}[a] \bigvee_{a \in Act}\{Z_{Q'} | Q \xrightarrow{a} Q'\}) \\
\text{iff [Definition of } \xrightarrow{a}\text{]} \\
P \in M_P(\bigwedge_{a \in Act}[a] \bigvee_{a \in Act, Q' : Q \xrightarrow{a} Q'} Z_{Q'})
\end{array}
$$

For the weak simulation we can give the following definition.

**Definition 2.28** *Given a finite-state process $P$, its characteristic formula with respect to strong and weak simulation is given by the closed list $D_P \downarrow Z_P$ where, respectively:*

**Strong** *for every $P' \in Der(P)$,*

$$Z_{P'} =_\nu \bigwedge_{a \in Act} ([a](\bigvee_{P'' : P' \xrightarrow{a} P''} Z_{P''}))$$

**Weak** *for every $P' \in Der(P)$,*

$$Z_{P'} =_\nu \bigwedge_{a \in Act} ([a](\bigvee_{P'' : P' \stackrel{\hat{a}}{\Rightarrow} P''} Z_{P''}))$$

As a matter of fact, we can follows the same reasoning made for strongly similar processes by considering that $(P, Q)$ are weakly similar, *i.e.*, are in $\mathcal{R}$, if and only if

$$\forall a \; P \xrightarrow{a} P' \quad \exists Q' \quad Q \stackrel{\hat{a}}{\Longrightarrow} Q' \wedge (P', Q') \in \mathcal{R}$$

Following a similar reasoning made in [105], it is possible to prove that the following proposition holds because of the definition of the characteristic formula does not depends on $Q$.

**Lemma 2.3** *Let $P$ and $Q$ be a finite-state process and let $\phi_{P,\prec}$ be the characteristic formula of the process $P$ with respect to strong simulation then:*

$$Q \prec P \Leftrightarrow Q \models \phi_{P,\prec}$$

*The same result holds also if we are considering the weak simulation as behavioral relation.*

*Proof*: see Appendix A.2.

□

**A timed setting** It is possible to characterize also processes in a timed setting. According to the definition of timed strong and weak simulation and bisimulation given in the previous section, it is possible to give the following definition.

**Definition 2.29** *([105]) Given a finite state process $P$, its* characteristic formula with respect to timed strong bisimulation *is given by the closed list $D_P \downarrow Z_P$ is defined by the following equation for every $P' \in Der(P)$, $\alpha \in Act_t$:*

$$Z_{P'} =_\nu \Big( \bigwedge_{\alpha \in Act_t; P'' : P' \xrightarrow{\alpha} P''} \langle \alpha \rangle Z_{P''} \Big) \wedge \Big( \bigwedge_{\alpha \in Act_t} ([\alpha] \Big( \bigvee_{P'' : P' \xrightarrow{\alpha} P''} Z_{P''} \Big) \Big) \Big)$$

Note that the presence of $tick$ actions does not influence the definition of the characteristic formula. For that reason, definitions 2.27 and 2.28 do not change in the timed setting.

The following lemmas hold.

**Lemma 2.4 ([116])** *Let $P_1$ and $P_2$ be two different finite-state processes. If $\phi_{P_2}$ is characteristic for $P_2$ then:*

1. *If $P_1 \approx_t P_2$ then $P_1 \models \phi_{P_2}$;*
2. *If $P_1 \models \phi_{P_2}$ and $P_1$ is finite-state process then $P_1 \approx_t P_2$.*

**Lemma 2.5** *Let $P$ and $Q$ be a finite-state process and let $\phi_{P,\preceq_t}$ be the characteristic formula of the process $P$ with respect to timed weak simulation then:*

$$Q \preceq_t P \Leftrightarrow Q \models \phi_{P,\preceq_t}$$

*Proof*: Since the definition of the characteristic formula with respect to simulation does not change by introducing the $tick$ action, the proof of this lemma follows the same step of the proof of Lemma 2.3 (see Appendix A.2).

□

### 2.3.4 Modeling web services through $CCS$

Here, we give some basic notions of service-oriented computing, an emerging paradigm that will serve to this thesis as one of the possible applications of the synthesis methodology. Upon a brief description of the general concepts of so called Web Services, we will show a useful mapping from their description languages to process algebra $CCS$.

Service-oriented computing is based on autonomous, platform-independent computational entities (called services) that can be described, published and categorized, and dynamically discovered and assembled for developing massively distributed, interoperable, evolvable systems and applications. The widespread success that this paradigm achieves today can be witnessed by the effort and resources that many large companies invest, to promote service delivery on a variety of computing platforms, mostly through the Internet in the form of Web Services. Tomorrow, the expectation is that there will be a plethora of new services supporting, *e.g.*, e-government, e-business, e-science. These will promote a rapid evolution of the Information Society.

The *World Wide Web Consortium*, $W3C$ for short, defines a *Web Service* as a software system designed to support interoperable *Machine to Machine* interaction over a network. Web Services are frequently just Web $API$s that can be accessed over a network, such as the Internet. They are executed on a remote system with respect to the system of the user that invokes that service. Eventually, the user gets the result of such an execution.

To sum up, Web Services are computational entities distributed on the web, whose main goal is to cooperate in order to work out simple or complex tasks.

#### Web Service languages: WSDL and BPEL

The $W3C$ definition of Web Service encompasses many different systems, but in common usage the term refers to clients and servers that communicate using XML (the *Extensible Markup Language*) messages following the $SOAP$ standard. $SOAP$ is a protocol for exchanging XML-based messages over computer networks.

XML is a general-purpose markup language. It is classified as an extensible language because it allows its users to define their own tags. Its primary purpose is to facilitate the sharing of structured data across different information systems, particularly via the Internet. It is used both to encode documents and serialize data.

WSDL is XML-based language that provides a model for describing Web Services. In particular, the format describes network services as a set of endpoints, or ports, operating on messages containing either document-oriented or procedure-oriented information.

The service is described, at an abstract level, in terms of the messages it sends and receives and, at a concrete level, defines details about protocols and data format specifications implementing operations for that particular service.

The BPEL language [4] has been introduced to describe business processes which manage the interaction of different Web Services, *i.e.*, it describes how Web Services can be composed and can cooperate one each other. Like WSDL, it is an XML-based language. It is layered on top of WSDL and it defines how to coordinate the interactions

| Primitive activities | |
|---|---|
| *Receive* | Accepts a message through the invocation of a specified operation by a partner |
| *Reply* | Sends a message as a response to a request previously accepted through a receive activity |
| *Throw* | Used when the process needs to signal a fault explicitly |
| *Invoke* | Used for invocation of a web service operation offered by a partner |
| *Link* | Defines a link of a flow; an activity within the flow can act as the source of a link or the target of a link |
| Structured activities | |
| *Flow* | Provides concurrency and synchronization (concurrent composition) |
| *While* | Supports repeated execution of a specified iterative activity; execution continues until the specified boolean condition no longer holds true |
| *Sequence* | Includes one or more activities to be executed sequentially, in the order in which the appear under this activity |
| *Pick* | Waits the appearance of one or more events and executes the activity associated with the event that emerged. Messages incoming or timer pass form the possible events |
| *Switch* | Supports conditional behavior by enabling specification of one or more case branches whose execution depends on a specified condition, and an optional else branch which gets executed if all cases fail their checks |

Table 2.8: BPEL relevant basic and structured activities.

between services. In this sense, a BPEL process definition provides and/or uses one or more WSDL interfaces, that are lists of message declarations and types, and it provides the description of the behavior and interactions of the services. Indeed, in WSDL no information is given on the sequence of messages sent and received by the service. This is one of the reason because BPEL has been adopted to describe the interactions between services.

In BPEL, specifications are classified as *basic* activities and *structured* activities (see Table 2.8).

Basic activities are sending and reception of a message, *e.g.*, receiving a request from a client, replying to the request, and also assigning data from one container to another, terminating the process, waiting for some period of time and doing nothing.

Structured activities define the control flow of the process. They include basic pro-

gramming constructs as *sequencing, loops* and *statements* of various kind, *e.g.*, :

$$\begin{array}{rl} \textit{sequencing}: & \langle \texttt{sequence} \rangle \\ & \quad \textit{activity}_1 \\ & \quad \textit{activity}_2 \\ & \quad \ldots \\ & \langle \backslash \texttt{sequence} \rangle \end{array}$$

that expresses a sequence of activity that are done successively.

$$\begin{array}{rl} \textit{while loops}: & \langle \texttt{while} \text{ condition=“bool-expr”} \rangle \\ & \quad \textit{activity} \\ & \langle \backslash \texttt{while} \rangle \end{array}$$

that describes that a certain activity is done while a the condition is verified.

$$\begin{array}{rl} \textit{switch statements}: & \langle \texttt{switch} \rangle \\ & \quad \langle \texttt{case condition} = \ldots \rangle \\ & \quad \langle \backslash \texttt{case} \rangle \\ & \quad \ldots \\ & \quad \langle \texttt{case condition} = \ldots \rangle \\ & \quad \langle \backslash \texttt{case} \rangle \\ & \langle \backslash \texttt{switch} \rangle \end{array}$$

that permits to chose which activity perform according to which case condition is verified.

BPEL also includes a structured activity called `pick`, that allows for nondeterministic selective communication:

$$\begin{array}{l} \langle \texttt{pick} \rangle \\ \quad \langle \texttt{onMessage} \ldots \rangle \\ \quad\quad \langle \texttt{invoke} \ldots \backslash \rangle \langle \backslash \texttt{onMessage} \rangle \\ \quad\quad \langle \texttt{onMessage} \ldots \rangle \\ \quad \langle \texttt{invoke} \ldots \backslash \rangle \langle \backslash \texttt{onMessage} \rangle \\ \langle \backslash \texttt{pick} \rangle \end{array}$$

This construct is similar to the nondeterministic choice construct of process algebras.

All the activities in BPEL are modeled as instantaneous, *i.e.*, they take no time. However, there are constructs modeling, *e.g.*, the elapsing of time: the `wait` activity allows a business process to specify a delay for a certain period of time or until a certain deadline is reached.

**From BPEL to $timedCCS$ process algebra.** The nature and features of BPEL suggest the use of a process algebra to formalize it. In the literature, several efforts have been done for relating BPEL with process algebras, see, *e.g.*, [9, 31, 123]. Indeed, process

algebras provide methodologies for the high-level description of interactions, communications, and synchronizations among processes, and this feature may be appealing for specifying interactions between web services, or reasoning on the specified system.

Referring to [31, 123], we recall and adapt here an encoding between BPEL and $timedCCS$.

The action notion of $timedCCS$ finds its equivalent in the *Receive*, *Reply* and *Invoke* basic activities of BPEL. As a matter of fact, an abstract action accompanied with a reception in $timedCCS$ may be expressed as a reception of a message using the receive activity in BPEL. On the other hand, an emission in $timedCCS$ corresponds in BPEL to the asynchronous invoke activity. At the abstract level, an emission followed immediately by a reception matches the BPEL synchronous invoke, performing two interactions (sending a request and receiving a response). On the other hand, the complementary reception/emission in $timedCCS$ is equivalent to a receive activity followed by a reply one in BPEL. Synchronized actions are equivalent to BPEL interactions. Hence BPEL services and $timedCCS$ processes correspond to each other.

Moreover, a system is described using a main behavior made up of instantiated processes composed in parallel and synchronizing together. As a matter of fact, the restriction operator is used in $timedCCS$ to make explicit the synchronization. Let us observe that the main specification $P_1 \| \ldots \| P_n$, that denotes a parallel composition, does not match with a BPEL process. Indeed the correspondence is between each process $P_1$ and BPEL service. Accordingly, the architecture of the specification is preserved.

Regarding the dynamic constructs,the sequence activity in BPEL matches the prefixing construct of $timedCCS$. The *nondeterministic choice* in $timedCCS$ can be seen as `sequence` and `pick` constructs in BPEL. In case of a deterministic choice described as a `switch` construct, we should use the same $timedCCS$ choice operator. An overall activity is completed when the end of its behavior is reached (no explicit construct unlike the termination denoted by $\mathbf{0}$ in $timedCCS$). Agent *recursion*, corresponding to the repetition of their behavior, could be represented using a `while` activity. To sum up in Table 2.9 there is the encoding proposed in [31, 123]. We stress that some notions that are present in $timedCCS$ do not directly appear in BPEL. This is the case of the $\tau$ action and of the restriction operator. Moreover, the behavior of the *idling* operator matches the BPEL `wait` activity. Furthermore, we can imagine that the names needed in the $timedCCS$ restriction set could be easily extracted from the WSDL files.

## 2.4 Compositional analysis

In this section we recall the theory on compositionality context developed in [77]. The problem under consideration is the following:

*What properties must the component of a combined system satisfy in order that the overall system satisfies a given specification.*

| BPEL | $timedCCS$ |
|---|---|
| `receive`, `reply`, `invoke` | actions(reply/receive) |
| `sequence` | sequence $\cdot$ |
| `pick` and `switch` | choice $+$ |
| interacting Web services | parallel composition $\|$ |
| interactions and `assign` | restriction $\backslash$ |
| end of the main `sequence` or `terminate` | termination $\mathbf{0}$ |
| new instantiation or `while` | recursive call |
| (internal) `assign`, (external) interactions | $\tau$-actions |
| `wait` | idling |

Table 2.9: Mapping between $timedCCS$ and WSDL/BPEL (extension of [31, 123] with time).

This kind of problem can be found, for instance, when a large system is developed. Since the implementation cannot be immediately extracted from the specification, the implementation phase consists of a large number of small refinements of the initial specification until, eventually, the implementation can be clearly identified.

In [77] the authors have developed a framework, based on the concept of *context*, in order to study this problem. The component should satisfy properties with the following features:

- properties should be as *weak* as possible, in order to not restrict too much the component implementation.
- properties should be *decomposable* into properties of subcomponents, thereby allowing to carry out independently the future refinement of the subcomponents.

### 2.4.1 Contexts

We present an operational theory of *context* in terms of *action transducers* as defined in [77]. We show that all $CCS$ operators can be described in this theory.

**Definition 2.30** *A* context *system $\mathcal{C}$ is a structure*

$$\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$$

*where $C_n^m$ is a set of $n$-to-$m$ contexts; $Act$ is a set of actions; $Act_0 = Act \cup \{0\}$ where $0 \notin Act$ is a distinguished no-action symbol, $Act_0^k$ is a tuple of $k$ actions $\in Act_0$, and $\rightarrow_{n,m} \subseteq C_n^m \times (Act_0^n \times Act_0^m) \times C_n^m$ is the transduction-relation for the $n$-to-$m$ contexts satisfying $(C, \overrightarrow{a}, \overrightarrow{0}, D) \in \rightarrow_{n,m}$ if and only if $C = D$ and $\overrightarrow{a} = \overrightarrow{0}$ for all contexts $C, D \in C_n^m$.*

For $(C, \overrightarrow{a}, \overrightarrow{b}, C') \in \rightarrow_{n,m}$ we usually write $C \overset{\overrightarrow{b}}{\overrightarrow{a}} C'$, leaving the indices of $\rightarrow$ to be determine by the context, and we interpret this as: *by consuming the action $\bar{a}$, the context $C$ can produce the actions $\overrightarrow{b}$ and change in $C'$.*

The operational semantics of contexts is consistent with the existing operational semantics of processes introduced in Section 2.3.1. For instance, let $C$ be a context $\in C_n^m$. If there are $n$ components $P_i$, $i = 1 \ldots n$, and $P_i \xrightarrow{a_i} P_i'$ is a transition of the component $P_i$, then $C$ can *consume* the actions $a_1 \ldots a_n$ while *producing* action $b_1 \ldots b_m$ and changing into $C'$. Thus, the context $C(P_1, \ldots, P_n)$ has the transition $C(P_1, \ldots, P_n) \xrightarrow[a_1 \ldots a_n]{b_1 \ldots b_m} C'(P_1', \ldots, P_n')$. Dually, any transition of $C(P_1, \ldots, P_n)$ ought to be derivable in this way.

In particular the set of 0-to-1 contexts $C_0^1$ are just processes and $C_n^1$ are normal $n$-hole contexts.

**Example 2.5** *$CCS$ process algebra (see [100]) can be seen as a context system with the following contexts:* prefix $a^* \in C_1^1$ *for* $a \in Act$, restriction $\backslash L \in C_1^1$ *where* $L \subseteq Act$. Choice *and* parallel *context* $+, \| \in C_2^1$*;* inactive $\mathcal{O}_n^m \in C_n^m$ *for any* $n$ *and* $m$*, with* $Nil$ *that denotes the context* $\mathcal{O}_0^1$*. There are also the* identity *context* $I_n \in C_n^n$ *and the* projection $\Pi_n^i \in C_n^1$*. The semantics definition of $CCS$ context is in Table 2.5.*

### Operations between contexts

Different operations between contexts can be defined. First of all, we show how the behavior of two contexts can be compared by recalling the following definition of *simulation* and *bisimulation* equivalence.

**Definition 2.31** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system. Then a $n$-to-$m$* simulation *$\mathcal{R}$ is a binary relation on $C_n^m$ such that, whenever $(C, D) \in \mathcal{R}$ and $\overrightarrow{a} \in Act_0^n$, $\overrightarrow{b} \in Act_0^m$, then the following holds:*

$$\text{if } C \overset{\overrightarrow{b}}{\overrightarrow{a}} C', \text{ then } D \overset{\overrightarrow{b}}{\overrightarrow{a}} D' \text{ for some } D' \text{ with } (C', D') \in \mathcal{R}.$$

*We write $C \prec D$ in case $(C, D) \in \mathcal{R}$ for some $n$-to-$m$ simulation $\mathcal{R}$.*

*A* bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations. We represent with $\sim$ the union of all the bisimulations.*

**Composition.** Contexts can be composed. $C(P_1, \ldots, P_n)$ is a composed context. In order to facilitate *composition*, it is allowed that contexts produce a number $m$ of actions, with respect to the consumption of $n$ actions, with, possibly, $n \neq m$. Moreover, in order to cater for *asynchronous* contexts, it is not required that all the components $P_1, \ldots, P_n$ contribute in a transition of the combined process $C(P_1, \ldots, P_n)$.

Inaction:

$$C \overset{\overrightarrow{0}}{\rightrightarrows} 0 \; C \text{ for all } C$$

Prefix:

$$a^* \overset{a}{\rightrightarrows} 0 \; I_1$$

Restriction:

$$\backslash L \overset{a}{\rightarrow} \backslash L \quad a \notin L$$

Choice:

$$(1) + \overset{a}{\overrightarrow{(a,0)}} \Pi_1^1 \quad (2) + \overset{a}{\overrightarrow{(0,a)}} \Pi_2^1 \text{ for } a \in Act$$

Projection:

$$\Pi_n^i \overset{a}{\overrightarrow{i(a)}} \Pi_n^i$$

Parallel:

$$(1) \parallel \overset{\tau}{\overrightarrow{(a,\bar{a})}} \parallel \quad (2) \parallel \overset{a}{\overrightarrow{(a,0)}} \parallel \quad (3) \parallel \overset{a}{\overrightarrow{(0,a)}} \parallel$$

Identity:

$$I_n \overset{\bar{a}}{\rightrightarrows} a \; I_n$$

where $i(a) \in Act_0^n$ with the $i^{th}$ component being $a$ and all the others being $0$.

Table 2.10: Semantics of $CCS$ context system.

**Definition 2.32** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system. A* composition *on $\mathcal{C}$ is a dyadic operation $\circ$ on contexts such that whenever $C \in C_n^m$ and $D \in C_m^r$ then $D \circ C \in C_n^r$. Furthermore, the transductions for a context $D \circ C$ with $C \in C_n^m$ and $D \in C_m^r$ are fully characterized by the following rule:*

$$\frac{C \overset{\overrightarrow{b}}{\rightrightarrows} a \; C' \quad D \overset{\overrightarrow{c}}{\rightrightarrows} b \; D'}{D \circ C \overset{\overrightarrow{c}}{\rightrightarrows} a \; D' \circ C'}$$

As a particular case of composition of context we can consider a unary context $C$ and a process $P$ and we may syntactically form the *combined process* $C(P)$ by substituting the free variable (normally denoted by []) in $C$ with $P$. Semantically, the behavior of $C(P)$ is derivable from the ones of $C$ and $P$. In particular, if $P \overset{a}{\longrightarrow} P'$ and $C$ has an $a$-consuming transduction $C \overset{b}{\rightrightarrows} a \; C'$, then the combined process $C(P)$ should have a $b$-transition to $C'(P')$. Extending the transition relation for processes such that $P \overset{0}{\longrightarrow} Q$ if

and only if $P = Q$, we have the following rule:

$$\frac{C \overset{b}{\overset{\rightarrow}{a}} C' \quad P \xrightarrow{a} P'}{C(P) \xrightarrow{b} C'(P')} \tag{2.1}$$

In particular, whenever $C \overset{b}{\overset{\rightarrow}{0}} C'$, *i.e.*, $C$ has a transduction which does not involve any consumption, we expect a transition $C(P) \xrightarrow{b} C'(P)$, with no effect on the inner process $P$.

**Product.** In order to represent a *partial specification* of a system, *i.e.*, a system with $n$ holes, we use an $n$-to-1 context $C \in C_n^1$. To allow the expansion of the $n$ holes to be carried out independently, an *independent combination* of $n$ contexts is defined as $D_1 \times \ldots \times D_n$.

**Definition 2.33** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system. A* product *on $\mathcal{C}$ is a dyadic operation $\times$ on contexts, such that whenever $C \in C_n^m$ and $D \in C_r^s$ then $C \times D \in C_{n+r}^{m+s}$. Furthermore, the transduction for a context $C \times D$ are fully characterized by the following rule:*

$$\frac{C \overset{\overrightarrow{b}}{\overset{\rightarrow}{a}} C' \quad D \overset{\overrightarrow{d}}{\overset{\rightarrow}{c}} D'}{C \times D \overset{\overrightarrow{b}\,\overrightarrow{d}}{\overrightarrow{a}\,\overrightarrow{c}} C' \times D'}$$

*where juxtaposition of vectors $\overrightarrow{a} = (a_1, \ldots, a_n)$ and $\overrightarrow{c} = (c_1, \ldots, c_r)$ is the vector $\overrightarrow{a}\,\overrightarrow{c} = (a_1, \ldots, a_n, c_1, \ldots, c_r)$.*

**Feed-back.** In order to deal with *recursion*, a construction of *feed-back* on contexts is defined, such that whenever $C \in C_n^n$ then $C^\dagger \in C_0^n$ with the behavioral equation $C^\dagger \sim C \circ C^\dagger$ being satisfied. Formally, we have the following definition.

**Definition 2.34** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system. A* feed-back *on $\mathcal{C}$ is a unary operation $^\dagger$ on contexts of $\mathcal{C}$ such that, whenever $C \in C_n^n$, then $C' \in C_0^n$. Furthermore, the transduction for a context $C^\dagger$ with $C \in C_n^n$ is fully characterized by the rule:*

$$\frac{C \overset{\overrightarrow{b}}{\overset{\rightarrow}{a}} C' \quad C^\dagger \xrightarrow{\overrightarrow{a}} D}{C^\dagger \xrightarrow{\overrightarrow{b}} C' \circ D}$$

For this operations on contexts the following result holds.

**Theorem 2.3 ([77])** *$\sim$ is preserved by composition, product and feed-back of context.*

By following a reasoning similar to the one are made in [77] to prove the previous theorem, it is possible to note that the same result holds also for $\prec$.

We conclude this section by recalling the following result:

Associativity:

$(A_\circ)\ E \circ (D \circ C) \sim (E \circ D) \circ C \qquad (A_\times)\ E \times (D \times C) \sim (E \times D) \times C$

Distributivity:

$(D_\circ^\times)\ (D \circ C) \times (D' \circ C') \sim (D \times D') \circ (C \times C') \qquad (D_\times^\dagger)\ (D \times E)^\dagger \sim D^\dagger \times E^\dagger$

Identity:

$(I)\ I_m \circ C \sim C \circ I_n \sim C$

Zero:

$(Z_\circ)\ \mathcal{O}_r^m \circ \mathcal{O}_n^r \sim \mathcal{O}_n^m \qquad (Z_\times)\ \mathcal{O}_n^m \times \mathcal{O}_r^s \sim \mathcal{O}_{n+s}^{m+r} \qquad (Z_\dagger)\ (\mathcal{O}_n^n)^\dagger \sim (I_n)^\dagger \sim \mathcal{O}_0^s$

$(ZZ)\ \mathcal{O}_m^n \circ C \circ \mathcal{O}_l^r \sim \mathcal{O}_l^n \quad (Z_0)\ \mathcal{O}_0^0 \times C \sim C \times \mathcal{O}_0^0 \sim C$

Projection:

$(P_d)\ \Pi_n^i \sim \mathcal{O}_{i-1}^0 \times I_1 \times \mathcal{O}_{n-i}^0 \qquad (P_\times)\ (I_m \times \mathcal{O}_s^0) \circ (C \times D) \circ (I_n \times \mathcal{O}_0^r) \sim C$

Fixedpoint:

$(F)\ C^\dagger \sim C \circ C^\dagger$

Table 2.11: Laws between operations on contexts.

**Theorem 2.4 ([77])** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system with composition, product, and feed-back. Then, the equivalences of Table 2.11 are universally valid.*

## 2.4.2 Property transformer

According to the definition in [77] it is possible to define the *property transformer* function $\mathcal{W}$ for contexts, as in Table 2.12. The property transformer is introduced in order to obtain the necessary and sufficient conditions that the unspecified part of the system has to satisfy. The following theorem holds.

**Theorem 2.5 ([77])** *Let $\mathcal{C} = (\langle C_n^m \rangle_{n,m}, Act, \langle \rightarrow_{n,m} \rangle_{n,m})$ be a context system. Let $\phi \in$ simultaneous $\mu$-calculus be a closed formula and let $C \in C_n^m$. Then, for any $Q \in C_0^n$, the following equivalence holds:*

$$C(Q) \models \phi \Leftrightarrow Q \models \mathcal{W}(C, \phi)$$

$$
\begin{array}{lcl}
\mathcal{W}(C, \mathbf{T}) & = & \mathbf{T} \\
\mathcal{W}(C, \mathbf{F}) & = & \mathbf{F} \\
\mathcal{W}(C, X) & = & X^C \\
\mathcal{W}(C, \phi_1 \wedge \phi_2) & = & \mathcal{W}(C, \phi_1) \wedge \mathcal{W}(C, \phi_2) \\
\mathcal{W}(C, \phi_1 \vee \phi_2) & = & \mathcal{W}(C, \phi_1) \vee \mathcal{W}(C, \phi_2) \\
\mathcal{W}(C, \langle \overrightarrow{b} \rangle \phi) & = & \bigvee_{C \overrightarrow{\overrightarrow{a}} D} \langle \overrightarrow{a} \rangle \mathcal{W}(D, \phi) \\
\mathcal{W}(C, [\overrightarrow{b}] \phi) & = & \bigwedge_{C \overrightarrow{\overrightarrow{a}} D} [\overrightarrow{a}] \mathcal{W}(D, \phi) \\
\mathcal{W}(C, \text{LET MAX } D \text{ IN } \phi) & = & \text{LET MAX } D^T \text{ IN } \mathcal{W}(C, \phi) \\
\mathcal{W}(C, \text{LET MIN } D \text{ IN } \phi) & = & \text{LET MIN } D^T \text{ IN } \mathcal{W}(C, \phi)
\end{array}
$$

where

$$D^T = \{X^C = \mathcal{W}(C, \phi) | C \in C_n^m, X = \phi \in D\}$$

Table 2.12: Definition of the property transformer $\mathcal{W}$.

In Table 2.13 there are a number of useful properties of the transformer $\mathcal{W}$ w.r.t the operations between contexts.

Here we recall some examples in order to explain how the property transformer works.

**Example 2.6** *We want to find the weakest requirement on the process $P$ such that the combined process $a^* \circ P$ presents an infinite $a$-computation. $a^*$ is a 1-to-1 context that can perform an action $a$ and then behaves as its internal component $P$. Exploiting Theorem 2.5, where $C = a^*$ and $\phi = \text{LET MAX } X = \langle a \rangle X \text{ IN } X$ we have that the requirement comes as follows:*

$$
\begin{array}{lcl}
\mathcal{W}(C, \phi) & = & \\
& = & \text{LET MAX} \left\{ \begin{array}{l} X^{a^*} = \mathcal{W}(C, \langle a \rangle X) \\ X^{I_1} = \mathcal{W}(I_1, \langle a \rangle X) \end{array} \right\} \text{ IN} X^{a^*} \\
& = & \text{LET MAX} \left\{ \begin{array}{l} X^{a^*} = \langle 0 \rangle X^{I_1} \\ X^{I_1} = \langle a \rangle X^{I_1} \end{array} \right\} \text{ IN} X^{a^*} \\
& = & \text{LET MAX} \left\{ \begin{array}{l} X^{a^*} = X^{I_1} \\ X^{I_1} = \langle a \rangle X^{I_1} \end{array} \right\} \text{ IN} X^{a^*} \\
& = & \phi
\end{array}
$$

*where $I_1$ is the identity function on a single action. Hence the component process $P$ must has an infinite $a$-computation.*

**Example 2.7** *We consider again the formula that expresses the Chinese Wall policy given in the Example 2.2. We consider a distributed system $S = \|(X_1, X_2)$. Hence we calculate*

Composition:
$$\mathcal{W}(C \circ D, \phi) \equiv \mathcal{W}(D, \mathcal{W}(C, \phi))$$

Identity:
$$\mathcal{W}(I_n, \phi) \equiv \phi$$

Monotonicity:
$$\frac{\phi \Rightarrow \psi}{\mathcal{W}(C, \phi) \Rightarrow \mathcal{W}(C, \psi)}$$

Bisimilarity:
$$\frac{C \sim D}{\mathcal{W}(C, \phi) \equiv \mathcal{W}(D, \phi)}$$

Feed-back:
$$\frac{\mathbf{T} \Rightarrow \mathcal{W}(C, \phi)}{C^{\dagger} \in \phi}$$

Induction-Principle:
$$\frac{\forall \psi \quad \psi \Rightarrow \mathcal{W}(C, \phi[\psi \backslash X])}{C^{\dagger} \in \text{LETMAX } X = \phi \text{IN } X}$$

Table 2.13: Properties of the property transformer $\mathcal{W}$.

$\phi' = \mathcal{W}(\|, \phi)$. *Let us consider the following abbreviation:*

$$W^{\|} = \mathcal{W}(\|, W) \qquad V^{\|} = \mathcal{W}(\|, V)$$
$$\phi'_1 = \mathcal{W}(\|, \phi_1) \qquad \phi'_2 = \mathcal{W}(\|, \phi_2)$$

*Then we obtain* $\phi' = \phi'_1 \vee \phi'_2$ *where* $\phi'_1$ *and* $\phi'_2$ *are the following:*

$$\phi'_1 = \text{LET MAX} W' = [(0, open_A)]W' \wedge [(open_A, 0)]W'$$
$$\wedge [(0, open_B)]\mathbf{F} \wedge [(open_B, 0)]\mathbf{F}) \text{IN} W'$$
$$\phi'_2 = \text{LET MAX} V' = [(0, open_B)]V' \wedge [(open_B, 0)]V'$$
$$\wedge [(0, open_A)]\mathbf{F} \wedge [(open_B, 0)]\mathbf{F} \text{IN} V'$$

**Partial evaluation function**

In [3], Andersen deals with the same problem. He has proposed the *partial model checking* mechanism in order to give a compositional method for proving properties of concurrent systems, *i.e.*, the task of verifying an assertion for a composite process is decomposed into verification tasks for the subprocesses. The *partial evaluation function* proposed by Andersen is very similar to the property transformer we have recalled in the previous section.

We decide to give here also the formulation of the *partial evaluation function* given in [3] for several reasons. First of all, its notation is more readable than the other one.

$$
\begin{array}{lcl}
(D\downarrow Z)//t & = & (D//t)\downarrow Z_t \\
\epsilon//t & = & \epsilon \\
(Z =_\sigma \phi D)//t & = & ((Z_s =_\sigma \phi//s)_{s\in Der(E)})(D)//t \\
Z//t & = & Z_t \\
\phi_1 \wedge \phi_2//s & = & (\phi_1//s) \wedge (\phi_2//s) \\
\phi_1 \vee \phi_2//s & = & (\phi_1//s) \vee (\phi_2//s) \\
[a]\phi//s & = & [a](\phi//s) \wedge \bigwedge_{s\xrightarrow{a}s'} \phi//s', \text{ if } a \neq \tau \\
\langle a\rangle\phi//s & = & \langle a\rangle(\phi//s) \vee \bigvee_{s\xrightarrow{a}s'} \phi//s', \text{ if } a \neq \tau \\
[\tau]\phi//s & = & [\tau](\phi//s) \wedge \bigwedge_{s\xrightarrow{\tau}s'} \phi//s' \wedge \bigwedge_{s\xrightarrow{a}s'} [\overline{a}](\phi\ //s') \\
\langle\tau\rangle\phi//s & = & \langle\tau\rangle(\phi//s) \vee \bigvee_{s\xrightarrow{\tau}s'} \phi//s' \vee \bigvee_{s\xrightarrow{a}s'} \langle\overrightarrow{a}\rangle(\phi//_{s'}) \\
\mathbf{T}//s & = & \mathbf{T} \\
\mathbf{F}//s & = & \mathbf{F}
\end{array}
$$

Table 2.14: Partial evaluation function for parallel operator.

Moreover, Andersen has studied system with only one unspecified component, and a relevant part of this thesis is about systems with only one unspecified component. Also, the use of partial model checking techniques allows to do model checking in an efficient way. On the other hand, we will use compositionality theory through contexts when we will deal with distributed systems, in which more than one component is unspecified.

The intuitive idea underlying the partial model checking is the following: proving that $P\|Q$ satisfies an equational $\mu$-calculus formula $\phi$ is equivalent to prove that $Q$ satisfies a modified specification $\phi_{//_P}$, where $//_P$ is the partial evaluation function for the parallel composition operator (see [3] or Table 2.14). The formula $\phi$ is specified by use the equational $\mu$-calculus. Hence, the behavior of a component is partially evaluated and the requirements are changed in order to respect this evaluation. The partial model checking function (also called partial evaluation function) for the parallel operator is given in Table 2.14.

In order to explain better how partial model checking function acts on a given equational $\mu$-calculus formula, we show the following example.

**Example 2.8** *Let $[\tau]\phi$ be the given formula and let $P\|Q$ be a process. We want to evaluate the formula $[\tau]\phi$ w.r.t. the $\|$ operator and the process $P$. The formula $[\tau]\phi_{//_P}$ is satisfied by $Q$ if the following three conditions hold at the same time:*

- *$Q$ performs an action $\tau$ going in a state $Q'$ and $P\|Q'$ satisfies $\phi$; this is taken into account by the formula*
  *$[\tau](\phi_{//_P})$.*

- *$P$ performs an action $\tau$ going in a state $P'$ and $P'\|Q$ satisfies $\phi$, and this is considered by the conjunction $\wedge_{P\xrightarrow{\tau}P'}\phi_{//_{P'}}$, where every formula $\phi_{//_{P'}}$ takes into account the behavior of $Q$ in composition with a $\tau$ successor of $P$.*

$$
\begin{array}{lcl}
(D\downarrow Z)//t & = & (D//t)\downarrow Z_t \\
\epsilon//t & = & \epsilon \\
(Z =_\sigma \phi D)//t & = & ((Z_s =_\sigma \phi//s)_{s\in Der(E)})(D)//t \\
Z//t & = & Z_t \\
\phi_1 \wedge \phi_2//s & = & (\phi_1//s) \wedge (\phi_2//s) \\
\phi_1 \vee \phi_2//s & = & (\phi_1//s) \vee (\phi_2//s) \\
\mathbf{T}//s & = & \mathbf{T} \\
\mathbf{F}//s & = & \mathbf{F} \\
[a]\phi//s & = & [a](\phi//s) \wedge \bigwedge_{s\xrightarrow{a}s'} \phi//s', \text{ if } a \neq \tau \\
[\tau]\phi//s & = & [\tau](\phi//s) \wedge \bigwedge_{s\xrightarrow{\tau}s'} \phi//s' \wedge \bigwedge_{s\xrightarrow{a}s'} [\overline{a}](\phi//s') \\
\langle a\rangle\phi//s & = & \langle a\rangle(\phi//s) \vee \bigvee_{s\xrightarrow{a}s'} \phi//s', \text{ if } a \neq \tau \\
\langle \tau\rangle\phi//s & = & \langle \tau\rangle(\phi//s) \vee \bigvee_{s\xrightarrow{\tau}s'} \phi//s' \vee \bigvee_{s\xrightarrow{a}s'} \langle \overrightarrow{a}\rangle(\phi//_{s'}) \\
[tick]\phi//s & = & \begin{cases} [tick]\phi//s' & s \xrightarrow{tick} s' \\ \mathbf{T} & otw \end{cases} \\
\langle tick\rangle\phi//s & = & \begin{cases} \langle tick\rangle\phi//s' & s \xrightarrow{tick} s' \\ \mathbf{F} & otw \end{cases}
\end{array}
$$

Table 2.15: Partial evaluation function for parallel operator of $timedCCS$.

- *the $\tau$ action is due to the performing of two complementary actions by the two processes. So for every $a$-successor $P'$ of $P$ there is a formula $[\overrightarrow{a}](\phi_{//_{P'}})$.*

In [3], the following lemma is given.

**Lemma 2.6** *Given a process $P\|Q$ (where $P$ is a finite-state process) and an equational specification $D \downarrow Z$ we have:*

$$P\|Q \models (D \downarrow Z) \textit{ iff } Q \models (D \downarrow X)_{//_P}$$

The reduced formula $\phi_{//P}$ depends only on the formula $\phi$ and on process $P$. No information is required on the process $Q$ which can represent a possible enemy.

Remarkably, this function is exploited in [3] to perform model checking efficiently, where both $P$ and $Q$ are specified. In our setting, the process $Q$ will be not specified. Thus, given a certain system $P$, it is possible to find the property that the enemy must satisfy to avoid a successful attack on the system. It is worth noticing that partial model checking function may be automatically derived from the semantics rules used to define a language semantics. Thus, the proposed technique is very flexible. According to [3], when $\phi$ is *simple*, *i.e.*, it is of the form $X$, $\mathbf{T}$, $\mathbf{F}$, $X_1 \wedge \ldots \wedge X_k \wedge [\alpha_1]Y_1 \wedge \ldots \wedge [\alpha_l]Y_l$, $X_1 \vee \ldots \vee X_k \vee \langle\alpha_1\rangle Y_1 \vee \ldots \vee \langle\alpha_l\rangle Y_l$, then the size of $\phi_{//P}$ is bounded by $|\phi| \times |P|$. Referring to [2], any assertion can be transformed to an equivalent simple assertion in linear time. Hence, we can conclude that the size of $\phi_{//P}$ is polynomial in the size of $\phi$ and $P$.

It is important to note that a lemma similar to Lemma 2.6 holds for each $CCS$ operators.

**Partial model checking in a timed setting.** According to [116], the partial model checking function can be extended to deal with system in a timed setting, *i.e.*, with respect to $timedCCS$ operators.

Referring to the semantics definition of $timedCCS$ given in Table 2.7, we can note that, by introducing the new $tick$ action, the semantics definition of the choice operators and of the parallel one have a new rule. Moreover, we have a new operator, the idling.

In Table 2.15, we recall the definition of the partial model checking function for parallel operator in a timed setting.

# Chapter 3

# Run-time monitors and enforcement techniques

In this chapter we present several techniques to enforce *security properties* that specify acceptable executions of programs. For example, a security property might concern *access control* that specifies what operations individuals can perform on objects, *information flow* that specifies what individuals can infer about objects from observing other aspects of the system behavior, or *availability* that ensures that information or resources are available when required.

According to [124], an *enforcement mechanism*, $EM$ for short, is a mechanism that works by monitoring a *target system*, that is the system we want to check, and terminating any execution that is about to violate the security policy being enforced. Class of $EM$ includes *security kernels*, *reference monitors*, and other operating systems and hardware-based enforcement mechanisms.

Here, we propose process algebra *controller operators* as mechanisms to enforce security properties at run time. As a matter of fact, our framework is based on process algebra (see Section 2.3), partial model checking (see Section 2.6) and *open system paradigm* suggested for the modeling and the verification of system, and here extended to deal with the synthesis problem (see Chapter 4). Using the open system approach we develop a theory to enforce security properties. Our goal consists in protecting the system against possible intruders. Indeed, we should check each process that could interact with the system, considering it as an intruder or a malicious agent, before executing it. If it is not possible, we have to find a way to guarantee that the whole system behaves correctly, even when there are intruders.

The technical proofs of the results in this chapter are in the Appendix A.1.

## 3.1 Specification and verification of secure systems

Following the approach proposed in [85, 89], we describe a methodology for the formal analysis of secure systems based on the concept of open systems and partial model

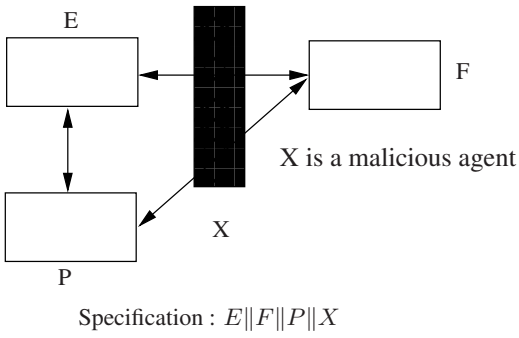



Figure 3.1: Graphical representation of a possible open system scenario.

checking techniques.

### 3.1.1 Open systems analysis for security

As reminded in the introduction, a system is *open* if it has some unspecified components. We want to make sure that the system with the unspecified component works properly, *e.g.*, by fulfilling a certain property. Thus, the intuitive idea underlying the verification of an open system is the following:

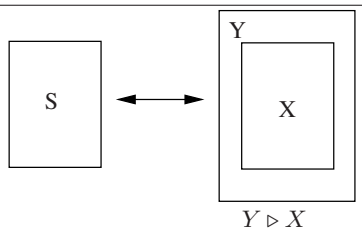
*An open system satisfies a property if and only if, whatever component is substituted to the unspecified one, the whole system satisfies this property.*

In the context of formal languages, an open system may be simply regarded as a term of this language which may contain "holes" (or placeholders). These are unspecified components. For instance $E\|(\_)$ and $E\|F\|(\_)$ may be considered as open systems.

**Example 3.1** *We suppose to have a system $S$ in which three processes $E$, $F$ and $P$ work in parallel. In order to be sure that $S$ works as we expected we have to consider that a possible malicious agent works in parallel with $E$, $F$ and $P$ as we can see in Fig. 3.1. In this case the possible intruder, here denoted by $X$, is able to interact with the other components in order to make the system unsafe. For that reason, instead to consider and analyze the system $S = E\|F\|P$, we study $S = E\|F\|P\|X$ and we require that $S$ is safe whatever the behavior of $X$ is.*

The main idea is that, when analyzing security-sensitive systems, neither the enemy's behavior nor the malicious users' behavior should be fixed beforehand. A system should be secure regardless of the behavior that the malicious users or intruders may have, which is exactly a *verification* problem of open systems. According to [85, 89], the problem that we want to study can be formalized as follows:

$$\text{For every component } X \quad S\|X \models \phi \tag{3.1}$$




Specification : $S \| Y \triangleright X$

Figure 3.2: A graphical representation of how a controller program $Y$ works.

where $X$ stands for a possible enemy, $S$ is the system under examination and $\phi$ is a (temporal) logic formula expressing the security property. It roughly states that the property $\phi$ holds for the system $S$, regardless of the unspecified component which may possibly interact with it. By using partial model checking we reduce such a verification problem as in Formula (3.1) to a validity checking problem as follows:

$$\forall X \quad S \| X \models \phi \quad \text{iff} \quad X \models \phi_{//_S} \tag{3.2}$$

In this way we find the sufficient and necessary condition on $X$, expressed by the logical formula $\phi_{//_S}$, such that the whole system $S \| X$ satisfies $\phi$ if ad only if $X$ satisfies $\phi_{//_S}$.

## 3.2 Process algebra controller operators

According to the Formula (3.2), in order to protect the system we should check each process $X$ before executing it. If it is not possible, we have to find a way to guarantee that the whole system behaves correctly. For that reason we develop *process algebra controller operators* that force the intruder to behave correctly, *i.e.*, referring to Formula (3.2), as prescribed by the formula $\phi_{//_S}$. We denote controller operators by $Y \triangleright X$, where $X$ is an unspecified component (*target*) and $Y$ is a *controller program*. The controller program is a process that controls $X$ in order to guarantee that a given security property is satisfied. Hence, we use controller operators in such a way the specification of the system becomes:

$$\exists Y \quad \forall X \quad \text{s.t.} \quad S \| (Y \triangleright X) \models \phi \tag{3.3}$$

By partially evaluating $\phi$ with respect to $S$ the Formula (3.3) is reduced as follows:

$$\exists Y \quad \forall X \quad Y \triangleright X \models \phi' \tag{3.4}$$

where $\phi' = \phi_{//_S}$.

In this way the behavior of the safe and known part of the system is considered directly into the formula $\phi'$. The problem described by the Formula (3.4) is about the target system

$X$ and the controller program $Y$. The controller program has to work only on $X$ and does not care about the rest of the system.

There exist other approaches that deal with the whole system of interest. Obviously, our approach differs from those, since we are able to control only $X$. This is, of course, an advantage because, often, not all the system needs to be checked, or it is simply not convenient to check it as a whole. Also, some components could be trusted and one would like to have a method to constrain only un-trusted ones (*e.g.*, downloaded applets). Similarly, it could not be possible to build a monitor for a whole distributed architecture, while it could be possible to have it for some of its components.

Among the existing approaches, this work of thesis inherits from the one based on the security automata of [19, 20, 124]. Security automata have been used to enforce a target to behave correctly, thus actuating as controllers. We will recall this in the next sections. Our work consists in doing the same by means of process algebras. This allows compositionality and it permits to use existing results on process algebras to do analysis, verification and synthesis of secure systems.

There is not a unique way to control a target system in order to enforce security properties. According to which properties the system has to satisfy and through the way it has to satisfy them, it is possible to use a controller operator instead of another. Indeed, it is possible to define several controller operators with different behaviors.

In the following we define some of them, by distinguishing between the controller operators that enforce *safety properties* and *information flow properties*. We start by recalling how security automata act as controllers, and then we map them at process algebras level.

### 3.2.1 Modeling security automata by controller operators

In [124], a *security automaton* is defined as a triple $(\mathcal{Q}, q_0, \delta)$ where $\mathcal{Q}$ is a set of states, $q_0$ is the initial one and $\delta : Act \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$, where $Act$ is a set of the actions, is the transition function. A security automaton processes a sequence $a_1 a_2 \ldots$ of actions. At each step only one action is considered and for each action we calculate the *global state* $Q'$ that is the set of the possible states for the current action, *i.e.*, if the automaton is checking the action $a_i$ then $Q' = \bigcup_{q \in Q'} \delta(a_i, q)$. If the automaton can make a transition on a given action, *i.e.*, $Q'$ is not empty, then the target is allowed to perform that step. The state of the automaton changes according to the transition rules. Otherwise, the target execution is terminated. Thus, at every step, it verifies if the action is in the set of the possible actions or not.

Successively, Ligatti *et al.* in [19, 20], starting from the definition of security automaton given by Schneider in [124], have defined four different kinds of deterministic security automata. To study their cost in term of time, we must consider how much the transition function costs. Since the security automata we consider are deterministic (thus $Q'$ would be either a singleton or empty), it is easy to understand that the cost in time is constant. Thus, given a sequence of $n$ actions, we need $O(n)$ to check whether this sequence is acceptable or not.

In the next section we recall the semantics definition of security automata *truncation, suppression, insertion, edit*, defined in [19, 20], and we will present how we model them by process algebra operators.

**Controller operators for enforcing safety properties**

We follow the approach given in [19] to describe the behavior of security automata *truncation, suppression, insertion, edit*.

These have a slightly different transition functions $\delta$, and these differences account for the variations in their expressive power. The exact specification of $\delta$ is part of the definition of each automaton. We use $\sigma$ to denote a sequence of actions, $\cdot$ for the empty sequence and $\tau$[1] to represent an internal action.

The execution of each different kind of security automata $\mathbf{K}$, with $\mathbf{K} \in \{T, S, I, E\}$, is specified by a labeled operational semantics. The basic single-step judgment has the form $(\sigma, q) \xrightarrow{a}_{\mathbf{K}} (\sigma', q')$ where $\sigma'$ and $q'$ denote, respectively, the actions sequence and the state after that the automaton takes a single step, $a$ denotes the action produced by the automaton. The single-step judgment can be generalized to a multi-step judgment $(\sigma, q) \stackrel{\gamma}{\Longrightarrow}_{\mathbf{K}}{}^{2} (\sigma', q')$, where $\gamma$ is a sequence of actions, as follows:

$$\frac{}{(\sigma, q) \stackrel{\cdot}{\Longrightarrow}_{\mathbf{K}} (\sigma, q)} \text{ (Reflex)}$$

$$\frac{(\sigma, q) \xrightarrow{a}_{\mathbf{K}} (\sigma'', q'') \quad (\sigma'', q'') \stackrel{\gamma}{\Longrightarrow}_{\mathbf{K}} (\sigma', q')}{(\sigma, q) \stackrel{a;\gamma}{\Longrightarrow}_{\mathbf{K}} (\sigma', q')} \text{ (Trans)}$$

where $a;\gamma$ means that the action $a$ and the sequence of actions $\gamma$ are performed sequentially.

We define four controller operators by showing their behavior thought semantics rules. We also prove for each operator that its behavior mimics the behavior of one of the security automata. Hence, in the following we recall the semantic definition of each security automaton, we show the controller operator by which we model it and, finally, we prove that they have the same behavior (for technical proofs see Appendix A.2).

**Truncation automaton.** The operational semantics definition of truncation automata given in [19, 20] is the following:

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

$$(\sigma, q) \xrightarrow{a}_{T} (\sigma', q') \qquad \text{(T-Step)}$$

[1] In [19] internal actions are denoted by $\cdot$. According to the standard notation of process algebras, we use $\tau$ to denote an internal action.

[2] Consider a finite sequence of visible actions $\gamma = a_1 \ldots a_n$. Here we use $\Rightarrow$ to denote automata computations. Before we use the same notation for process algebra computations. The meaning of the symbol will be clear from the context.

otherwise

$$(\sigma, q) \xrightarrow{\tau}_T (\cdot, q) \qquad \text{(T-Stop)}$$

We denote with $E$ the controller program and with $F$ the target. We work, without loss of generality, under the additional assumption that $E$ and $F$ never perform the internal action $\tau$ since truncation automata do not consider internal action. We define the controller operators $\triangleright_T$ as follows:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright_T F \xrightarrow{a} E' \triangleright_T F'}$$

This operator models the truncation automaton that is similar to Schneider's automaton (when considering only deterministic automata, *e.g.*, see [19, 20]). Its semantics rule states that if $E$ and $F$ perform the same action, thus such action is allowed. Hence, the controlled process $E \triangleright_T F$ performs the action $a$, otherwise it halts. It is easy to note that this operator is similar to the parallel operator of $CSP$ process algebra (see [67]).

It is also important to note that this operator is a monitor. As a matter of fact it is not able to enforce any security properties. On the other hand, it guarantees that a given security property will not be violated.

We give some notation. The $CCS$ choice operators on an arbitrary number of processes is denoted as $\sum_{a \in Act}$. For instance, $E = \sum_{act \in \{a,b,c\}} act.\mathbf{0}$ means that $E = a.\mathbf{0} + b.\mathbf{0} + c.\mathbf{0}$.

**Proposition 3.1** *Let*

$$E^q = \sum_{a \in Act} \begin{cases} a.E^{q'} & \textit{iff} \quad \delta(a,q) = q' \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of a* truncation automaton *$(\mathcal{Q}, q_0, \delta)$ is also derivable from $E^q \triangleright_T F$ and vice-versa.*

**Example 3.2** *Let us consider the system $S = a.b.\mathbf{0}$ and consider the following equational definition $\phi = [a][c]\mathbf{F}$. It asserts that after every action $a$, an action $c$ cannot be performed. Let $Act = \{a, b, c, \tau, \bar{a}, \bar{b}, \bar{c}\}$ be the set of actions.*

*In this case, by using the operator $\triangleright_T$, we are able to halt the execution of the system if, after an action $a$, it tries to perform an action $c$.*

**Suppression automaton.** It is defined as $(\mathcal{Q}, q_0, \delta, \omega)$ where $\omega : Act \times \mathcal{Q} \to \{-, +\}$ indicates whether or not the action in question should be suppressed (-) or emitted (+). Its semantics definition is the following:

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

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

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

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

otherwise

$$(\sigma, q) \xrightarrow{\tau}_S (\cdot, q) \qquad \text{(S-Stop)}$$

We denote with $E$ the controller program and with $F$ the target. Again, we suppose that $E$ and $F$ never perform the internal action $\tau$ since suppression automata do not consider internal action. We define the controller operators $\triangleright_S$ as follows:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright_S F \xrightarrow{a} E' \triangleright_S F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \triangleright_S F \xrightarrow{\tau} E' \triangleright_S F'}$$

where $-a$ is a control action not in $Act$ (so it does not admit a complementary action). As for the truncation automaton, if $F$ performs the same action performed by $E$ also $E \triangleright_S F$ performs it. On the contrary, if $F$ performs an action $a$ that $E$ does not perform and $E$ performs the control action $-a$ then $E \triangleright_S F$ performs the action $\tau$ that *suppresses* the action $a$, *i.e.*, $a$ becomes not visible from external observation. Otherwise, $E \triangleright_S F$ halts.

**Proposition 3.2** *Let* $E^{q,\omega} =$

$$\sum_{a \in Act} \begin{cases} a.E^{q',\omega} & \textit{iff} \quad \omega(a,q) = + \textit{ and } \delta(a,q) = q' \\ -a.E^{q',\omega} & \textit{iff} \quad \omega(a,q) = - \textit{ and } \delta(a,q) = q' \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of a* suppression automaton $(\mathcal{Q}, q_0, \delta, \omega)$ *is also derivable from $E^{q,\omega} \triangleright_S F$ and vice-versa.*

**Example 3.3** *Referring to the Example 3.2, we note that by using the operator $\triangleright_S$ we are able to check if the system tries to perform an action $c$ after the execution of an action $a$ and, consequently, it suppresses $c$.*

**Insertion automaton.** It is defined as $(\mathcal{Q}, q_0, \delta, \gamma)$ where $\gamma : Act \times \mathcal{Q} \to Act \times \mathcal{Q}$ that specifies the insertion of an action into the sequence of actions of the program.

In [19, 20], the automaton inserts a finite sequence of actions instead of only one action. By using $\gamma$, it controls if a wrong action is performed. If it happens, it inserts a finite sequence of actions, hence a finite number of intermediate states. Without loss of generality, here we consider that it inserts only one action. In this way we openly consider all the intermediate states. Note that the domain of $\gamma$ is disjoint from the domain of $\delta$, in order to have a deterministic automaton:

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

$$(\sigma, q) \xrightarrow{a}_I (\sigma', q') \qquad \text{(I-Step)}$$

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

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

otherwise

$$(\sigma, q) \xrightarrow{\tau}_I (\cdot, q) \qquad \text{(I-Stop)}$$

We denote with $E$ the controller program and with $F$ the target. $E$ and $F$ never perform the internal action $\tau$ since the insertion automata do not consider internal action. We define the controller operators $\triangleright_I$ as follows:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright_I F \xrightarrow{a} E' \triangleright_I F'} \qquad \frac{E \xrightarrow{q} \not\rightarrow E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \triangleright_I F \xrightarrow{b} E' \triangleright_I F}$$

where $+a$ is an action not in $Act$. If $F$ performs an action $a$ that also $E$ can perform, the whole system makes this action. If $F$ performs an action $a$ that $E$ does not perform and $E$ detects it by performing a control action $+a$ followed by an action $b$, then the whole system performs $b$. It is possible to note that in the description of insertion automata in [19] the domains of $\gamma$ and $\delta$ are disjoint. In our case, this is guaranteed by the premise of the second rule, *i.e.*, $E \xrightarrow{a} \not\rightarrow E'$, $E \xrightarrow{+a.b} E'$. In fact, if a pair $(a,q)$ is not in the domain of $\delta$ and it is in the domain of $\gamma$, it means that, in the state $q$, $a$ actions cannot be performed. Thus, in order to change state, an action different from $a$ must be performed. It is important to note that the controller is able to insert new actions, but it is not able to suppress any action performed by $F$.

**Proposition 3.3** *Let* $E^{q,\gamma} =$

$$\sum_{a \in Act} \begin{cases} a.E^{q',\gamma} & \textit{iff} \quad \delta(a,q) \\ +a.b.E^{q',\gamma} & \textit{iff} \quad \gamma(a,q) = (b,q') \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of an* insertion automaton $(\mathcal{Q}, q_0, \delta, \gamma)$ *is also derivable from $E^{q,\gamma} \triangleright_I F$ and vice-versa.*

**Example 3.4** *Let us consider again the same system and the same property of the Example 3.2. Applying the operator $\triangleright_I$ we obtain the same behavior of the $\triangleright_T$ operator, because there is no action to insert. The only possibility is to halt the system execution.*

**Edit automaton.** It is defined as $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$, where $\gamma : Act \times \mathcal{Q} \rightarrow Act \times \mathcal{Q}$ specifies the insertion of a finite sequence of actions into the program's actions sequence and $\omega : Act \times \mathcal{Q} \rightarrow \{-,+\}$ indicates whether or not the action in question should be suppressed (-) or emitted (+). $\omega$ and $\delta$ have the same domain, while the domain of $\gamma$ is disjoint from the domain of $\delta$, in order to have a deterministic automaton:

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

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

We denote with $E$ the controller program and with $F$ the target. $E$ and $F$ never perform the internal action $\tau$ since the edit automata do not consider internal action. In order to do insertion and suppression together, we define the following controller operator $\triangleright_E$. Its rules are the union of the rules of the $\triangleright_S$ and $\triangleright_I$.

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright_E F \xrightarrow{a} E' \triangleright_E F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \triangleright_E F \xrightarrow{\tau} E' \triangleright_E F'}$$

$$\frac{E \overset{a}{\not\longrightarrow} E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \triangleright_E F \xrightarrow{b} E' \triangleright_E F}$$

This operator combines the power of the previous ones.

**Proposition 3.4** *Let*

$$E^{q,\gamma,\omega} = \sum_{a \in Act} \begin{cases} a.E^{q',\gamma,\omega} & \textit{iff} \quad \delta(a,q) = q' \textit{ and } \omega(a,q) = + \\ -a.E^{q',\gamma,\omega} & \textit{iff} \quad \delta(a,q) = q' \textit{ and } \omega(a,q) = - \\ +a.b.E^{q',\gamma,\omega} & \textit{iff} \quad \gamma(a,q) = (b,q') \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of an* edit automaton $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$ *is also derivable from $E^{q,\gamma,\omega} \triangleright_E F$ and vice-versa.*

It is important to note that we have introduced the control action $-a$ in the semantics of $\triangleright_S$ and the control action $+a$ in the semantics of $\triangleright_I$ in order to find operators that are as similar as possible to suppression and insertion automata, respectively.

**Example 3.5** *Let us consider again the same system and the same property of the Example 3.2. Applying the operator $\triangleright_E$ we obtain the same behavior of the $\triangleright_S$ operator. Hence, the action $c$ is suppressed and the execution ends correctly.*

**Timed setting**

Note that as far as safety properties are concerned, Section 3.2.1, a timed setting does not influence the semantics definition of the operators $\triangleright_{\mathbf{K}}$. As a matter of fact, the $tick$ action will be performed when both the controller program and the target agree to perform it. Hence

$$\frac{E \xrightarrow{tick} E' \quad F \xrightarrow{tick} F'}{E \triangleright_{\mathbf{K}} F \xrightarrow{tick} E' \triangleright_{\mathbf{K}} F'}$$

It is easy to note that it is exactly the first rule of all the four operators in which, instead of a generic action $a \in Act$ we consider the action $tick$. Since, when we are working in a timed setting we consider $Act_t = Act \cup \{tick\}$ as set of actions, as we have defined in Section 2.3.2, the semantics definition of the four operators does not change.

### 3.2.2 Controller operators for enforcing information flow properties

*Information flow properties* are a particular class of security properties which aim at controlling the way information may flow among different entities. They have been first proposed as a mean to ensure confidentiality, in particular to verify if access control policies are sufficient to guarantee the secrecy of (possibly classified) information. Indeed, even if access control is a well studied technique for system security, it is not trivial to find an access control policy which guarantees that no information leak is possible.

In the literature, there are many different security definitions reminiscent of the information flow idea, each based on some system model (see [49, 51, 81, 121]). Here, we consider the notion of *Bisimulation Non Deducibility on Compositions*, $BNDC$ for short, proposed in [49, 51] as a generalization of the classical idea of *Non-Interference* [60] to nondeterministic systems: Low level users cannot infer the behavior of high level users from the system because for low level users the system appears always the same.

In the following we recall the definition of non interference as $BNDC$ property proposed by Focardi and Gorrieri [48, 49, 50, 51]. Moreover we recall the analysis method for this class of property based on *open system* paradigm proposed by Martinelli in [52, 89, 116].

**Information flow properties**

To describe information flow properties, we can consider two users, *High* and *Low*, interacting with the same computer system. We wonder if there is any flow of information from *High* to *Low*.

In [48, 49, 50, 51], Focardi and Gorrieri propose a family of information flow security properties called *Non Deducibility on Compositions* ($NDC$, for short). Intuitively, a system is $NDC$ if, by *interacting* with every possible high level user, this always *appears* the same to low level users. No information at all can be deduced by low level users.

The underlying idea of $NDC$ for ensuring non interference between high users and low users is that the system behavior perceived by low users must be invariant with respect

to the composition with every high user. Hence, there is no possibility of establishing a communication (*i.e.*, sending information); intuitively, it is like a medium where the same signal is always present. In terms of a generic language for the description of systems, where $\|$ stands for the composition operator and $\equiv$ for the equivalence relation, we have:

$$\forall \Pi \in \text{High users } E\|\Pi \equiv E \quad \text{with respect to } Low \text{ users}$$

This property can be instantiated by assuming different notions of composition and equivalence. The method presented in [116] is suitable to manage various composition operators and equivalence criteria. In terms of $CCS$ parallel composition operator and *bisimulation* equivalence, we have the definition of $BNDC$ (*Bisimulation based Non Deducibility on Compositions*).

**Definition 3.1** *Let $\mathcal{E}_H = \{\Pi \mid Sort(\Pi) \subseteq H \cup \{\tau\}\}$ be the set of High users. $E \in BNDC$ if and only if $\forall \Pi \in \mathcal{E}_H$ we have $(E\|\Pi)\backslash H \approx E\backslash H$.*

Let us show how $BNDC$ works through some simple examples.

**Example 3.6** *The simplest case of flawed process is $E = h.\bar{l}.\mathbf{0}$ where $h$ is high and $l$ is low. The process $E$ accepts the high level input $h$ and, only after such an input is receives, it gives $\bar{l}$ as output. As a consequence, a low level user knows that $h$ has been performed by just observing the output $\bar{l}$. This system is not $BNDC$. It is sufficient to consider $\Pi = \bar{h}.\mathbf{0}$ and observe that $(E\|\Pi)\backslash \approx \bar{l}.\mathbf{0}$ while $E\backslash H \approx \mathbf{0}$. Thus, $E\backslash H \not\approx (E\|\Pi)\backslash H$. $E$ can be made secure by letting $\bar{l}$ be also executed independently from $h$ as in $E' = h.\bar{l}.\mathbf{0} + \bar{l}.\mathbf{0}$. It is easy to prove that $E'$ is $BNDC$.*

By using the characteristic formula $\phi$ of the process $E\backslash H$ (see Section 2.3), we may express information flow property in a logical way as follows:

$$E \in BNDC \text{ if and only if } \forall \Pi \in \mathcal{E}_H \quad (E\|\Pi)\backslash H \models \phi \tag{3.5}$$

By using the partial model checking function, the property 3.5 turns out to be equivalent to:

$$E \in BNDC \text{ if and only if } \forall \Pi \in \mathcal{E}_H \quad \Pi \models \phi' \tag{3.6}$$

where $\phi'$ is the formula obtained from $\phi$ after the partial evaluation with respect to the process $E$ (and the restriction operator). Thus, due to the decidability of the validity problem for $\mu$-calculus we have that $BNDC$ for finite-state processes is decidable [52, 89].

### Enforcing $BNDC$ properties

The *open systens* paradigm is used also for enforcing $BNDC$ properties. In this case, the unspecified component will be the high level component.

According to Statement 3.5, we would like to guarantee that:

$$\forall X (S\|X)\backslash H \models \phi \tag{3.7}$$

where $S$ is the system, $X$ is the unspecified component, *e.g.*, a downloaded mobile agent, and $H = Sort(X)$.

We provide a framework for defining new operators, say $Y \triangleright X$, that can permit to control the behavior of the component $X$, given the behavior of a controller program $Y$.

$$\exists Y \quad \forall X \quad S \| (Y \triangleright X) \backslash H \models \phi \tag{3.8}$$

We use the partial model checking approach and we focus on the properties that the controller program $Y$ has to enforce.

$$\exists Y \quad \forall X \quad Y \triangleright X \models \phi_{S,\backslash H} \tag{3.9}$$

**Examples of controller operators** Let $E$ and $F$ be two processes, and let $a \in Act$ be an action. We define a new operator $\triangleright'$ (*controller operator*) by these two rules:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F'} \qquad \frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F} \tag{3.10}$$

This operator forces the system to make always the right action also if we do not know what action the agent $F$ is going to perform.

We can define other controller operators as follows.

The controller $\triangleright''$ have two rules:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F'} \qquad \frac{E \xrightarrow{a} E' \; F \overset{a}{\not\rightarrow}}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F} \tag{3.11}$$

This controller is the most complete: If $F$ does not have a correct behavior (*i.e.*, , it does not perform the right action $a$), the process $E$ takes care of it, and performs that action. Thus, the system maintains a correct behavior.

### Timed setting

Now, we propose controller operators for enforcing information flow in a timed setting. The introduction of the elapsing of time presents some difficulties when enforcing this kind of properties.

Let $tBNDC$ be the *Bisimulation based Non Deducibility on Compositions* property in a timed setting (see [49]).

In the $timedCCS$ model we cannot consider all the high processes for the interaction with the system. Indeed, we have to restrict the set of the admissible $High$ users. We consider *weakly time alive* processes that can perform only action in $H \cup \{\tau, tick\}$. and we denote by $\mathcal{E}_H$ the set of such processes.

This restriction is made because a process $\Pi$ that is not weakly time alive may prevent time from elapsing when composed in parallel with some system $E$. Indeed, in a compound process, time can pass if and only if all components let it pass. Hence, a high

user which is not weakly time alive could block the time flow also for low users and we certainly want to avoid this unrealistic (and undesirable) possibility. The *tBNDC* property in timed $CCS$ can be thus defined as follows.

**Definition 3.2** *$E \in$ tBNDC if and only if $\forall \Pi \in \mathcal{E}_H$ we have $(E\|\Pi)\backslash H \approx_t E\backslash H$.*

Due to the presence of the universal quantification, *tBNDC* is not very easy to check.

Let $\mathcal{H}$ be the set of high users that are composed with the system when it is checked (as done in [89]). Under certain constraints on the set $\mathcal{H}$, we can provide a method for reducing the verification of $tBNDC^{\mathcal{H}}$ membership to a validity problem in equational $\mu-$calculus, where by $tBNDC^{\mathcal{H}}$ we denoted $tBNDC$ for processes in $\mathcal{H}$,

**Definition 3.3** *$E \in tBNDC^{\mathcal{H}}$ if and only if*

$$\forall \Pi \in \mathcal{H}\ (E\|\Pi)\backslash H \approx_t E\backslash H$$

By using the characteristic formula for $\approx_t$ of $E\backslash H$, we obtain the following characterization: [3]

$$E \in tBNDC^{\mathcal{H}} \text{ if and only if } \forall \Pi \in \mathcal{H}\ (E\|\Pi)\backslash H \models \phi_{\approx_t, E\backslash H} \tag{3.12}$$

Now, we can apply the partial evaluation function with respect to $E, \backslash H$ to the formula $\phi_{\approx_t, E\backslash H}$ by getting a formula $\phi'$. Then the previous equation is equivalent to check that every process in $\mathcal{H}$ satisfies $\phi'$. Indeed, the behavior of $E$ has been evaluated and encoded in the formula $\phi'$. Thus:

$$E \in tBNDC^{\mathcal{H}} \text{ if and only if } \forall \Pi \in \mathcal{H} \quad \Pi \models \phi'$$

We expect to have decidability results only if we restrict ourselves to finite-state systems (see [86]). Let $fs = \{E | Der(E) \text{ is finite}\}$ be the set of finite state processes. We also require that the set $\mathcal{L}$ of visible actions is finite. If the membership in $\mathcal{H}$ can be defined by a formula $\phi''$ then we obtain that the previous problem is equivalent to:

$$E \in tBNDC^{\mathcal{H}} \text{ if and only if } \forall \Pi \in \mathcal{H} \quad \Pi \models \phi'' \Rightarrow \phi'$$

The validity problem for this logic may be shown to be decidable for finite-state processes by using the same proof techniques of [128].

**Enforcing $tBNDC$ properties.** Let $E$ and $F$ be two processes. We define the controller operator $\triangleright'$ as follows.

$$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\alpha} F'}{E \triangleright' F \xrightarrow{\alpha} E' \triangleright' F'} \quad \alpha \neq \tau \qquad \frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F} \qquad \frac{F \xrightarrow{\tau} F'}{E \triangleright' F \xrightarrow{\tau} E \triangleright' F'}$$

This operator forces the system to make always the right action also if we do not know what action $F$ is going to perform. Whereas we are interested in the observational equivalence between processes, $F$ can also perform the action $\tau$. Under this hypothesis and

[3] Actually, this is true only if we consider finite-state processes.

the additional one that $F$ is weakly time alive, this controller operator is able to wait an action of a possible intruder, then, after that timeout expires, performs the right action. It is possible to note that $\triangleright'$ is $tick$-deterministic (the action $tick$ can be performed only if the first rule can be applied). Another controller operator $\triangleright''$ could be defined as follows.

$$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\alpha} F'}{E \triangleright'' F \xrightarrow{\alpha} E' \triangleright'' F'} \quad \alpha \neq \tau \qquad \frac{E \xrightarrow{a} E' \quad F \overset{a}{\not\rightarrow} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F} \qquad \frac{F \xrightarrow{\tau} F'}{E \triangleright'' F \xrightarrow{\tau} E \triangleright'' F'}$$

This operator looks at the action performed by $F$ and, if $E$ and $F$ perform the same action $\alpha$, then the whole system performs it. On the contrary, the whole system performs the action performed by $E$. The $\tau$ action can be always performed by both of the processes. This controller is $tick$-deterministic.

### 3.2.3 Controller contexts for enforcing security properties in a distributed system

Following the same approach based on *open system* here we present how and when it is possible to enforce security properties in a distributed system by using centralized/ decentralized monitors. As a matter of fact we consider a partially specified systems in which more than one component is unspecified. We model it as a context in $C_n^1$ where $n$ is the number of unspecified components of the system.

We would like to guarantee that a given partially specified system is secure, *i.e.*, satisfy a given property $\phi$.

$$\forall X_1, \ldots, X_n \quad C(X_1, \ldots, X_n) \models \phi \tag{3.13}$$

We introduce *controller context*, denoted by $\triangleright \in C_{2n}^n$, that forces the system to behave correctly, *i.e.*, as prescribed by $\phi$. It acts on two components, in particular it combines a *controller program* $Y \in C_0^n$ and an unknown component $X \in C_0^n$ in such a way $Y$ forces $X$ to behave correctly.

### 3.2.4 Controller context for safety properties

It is possible to give several semantics definitions for $\triangleright$. It depends on which kind of properties we are going to enforce or in which way we want to enforce them (*e.g.*, [92, 93, 98]). Here we aim to define controller contexts for enforcing safety properties. Hence, referring to the Section 3.2.1, we present a possible variant of the semantics definition of controller operator $\triangleright_{\mathbf{K}}$ in order to deal with contexts. For simplicity we denote the controller contexts that we define below with the same symbols, $\triangleright_T$, $\triangleright_S$, $\triangleright_I$ and $\triangleright_E$, used in Section 3.2.1.

We define the controller context $\triangleright_T(Y, X)$ by the following semantics rule:

$$\frac{Y \xrightarrow{\vec{b}} Y' \quad X \xrightarrow{\vec{b}} X'}{\triangleright_T(Y, X) \xrightarrow{(\vec{b}, \vec{b})} \triangleright_T(Y', X')} \tag{3.14}$$

This means that $\triangleright_T$ works in such a way if $Y$ and $X$ do not perform the same actions than the execution halts.

The semantics definition of controller contexts $\triangleright_S(Y, X)$ is the following:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_S(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_S(Y',X')} \qquad \frac{Y \xrightarrow{\overrightarrow{-a}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_S(Y,X) \xrightarrow{(\overrightarrow{-a},\overrightarrow{a})} \triangleright_S(Y',X')} \tag{3.15}$$

where $\overrightarrow{-a}$ is a control vector of actions not in $Act_0^n$ (so it does not admit a complementary action). As for $\triangleright_T(Y, X)$, if $X$ performs the same vector of actions performed by $T$ also $\triangleright_S(Y, X)$ performs it. On the contrary, if $X$ performs $\overrightarrow{a}$ that $Y$ does not perform and $Y$ performs the control vector of actions $\overrightarrow{-a}$ then $\triangleright_S(Y, X)$ performs $\overrightarrow{\tau}$ that *suppresses* $\overrightarrow{a}$, *i.e.*, $\overrightarrow{a}$ becomes not visible from external observation. Otherwise, $\triangleright_S(Y, X)$ halts.

We define the controller context $\triangleright_I$ as follows:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_I(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_I(Y',X')} \qquad \frac{Y \not\xrightarrow{\overrightarrow{a}} Y' \quad Y \xrightarrow{\overrightarrow{+a.b}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_I(Y,X) \xrightarrow{(\overrightarrow{+a.b},\overrightarrow{a})} \triangleright_I(Y',X')} \tag{3.16}$$

where $\overrightarrow{+a.b}$ is a vector of action not in $Act_0^n$. We use this notation because we want to consider as a unique step that before performing the vector of action $\overrightarrow{b}$, $Y$ performs a control vector of action $\overrightarrow{+a}$. Informally, the semantics of $\triangleright_I$ means that, if $X$ performs an action $\overrightarrow{a}$ that also $Y$ performs, the whole system makes this action. On the contrary, if $Y$ detects that $X$ is going to perform a forbidden action by performing a control action $+a$ followed by an action $b$, then the whole system performs $b$.

The last controller context we define is $\triangleright_E(Y, X)$. Its rules are the union of the rules of the $\triangleright_S(Y, X)$ and $\triangleright_I(Y, X)$.

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_E(Y',X')} \qquad \frac{Y \xrightarrow{\overrightarrow{-a}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{-a},\overrightarrow{a})} \triangleright_E(Y',X')} \tag{3.17}$$

$$\frac{Y \not\xrightarrow{\overrightarrow{a}} Y' \quad Y \xrightarrow{\overrightarrow{+a.b}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{+a.b},\overrightarrow{a})} \triangleright_E(Y',X')}$$

This controller context combines the power of the two previous ones.

### Two approaches for enforcing safety properties in a distributed system

A controller context can be applied in several ways in order to solve the problem in Formula (3.13). Here we present two possible approaches: A centralized and a decentralized

method. In the following we use the symbol $\triangleright$ in order to represent a generic controller contexts. As a matter of fact, the following theory is not related to a particular controller context.

**Centralized method.** We use a unique controller program $Y$ that enforces $\phi$ by monitoring a unique unspecified component obtained by considering the product of all unspecified components of the systems. In this case the Formula (3.13) becomes:

$$\exists Y \quad \forall X_1, \ldots, X_n \quad C(\triangleright(Y, X_1 \times \ldots \times X_n)) \models \phi \tag{3.18}$$

where, being each $X_i$ is in $C_0^1$, $X_1 \times \ldots \times X_n$ is in $C_0^n$. First of all, by applying the property transformer function, we find the weakest condition the unknown component has to satisfy in order to guarantee the whole system satisfies $\phi$. Hence the problem of Formula (3.18) becomes as follows:

$$\exists Y \quad \forall X_1, \ldots, X_n \quad \triangleright (Y, X_1 \times \ldots \times X_n) \models \phi' \tag{3.19}$$

where $\phi' = \mathcal{W}(C, \phi)$.

**Decentralized method.** We use several controller programs $Y_i$, one for each unspecified components $X_i$ of the system. Hence our main problem can be formalized as follows:

$$\exists Y_1, \ldots, Y_n \; \forall X_1, \ldots, X_n \; C(\triangleright(Y_1, X_1), \ldots, \triangleright(Y_n, X_n)) \models \phi \tag{3.20}$$

In this case all $Y_i$ are processes. By exploiting the property transformer on context, we can reduce the previous problem as follows:

$$\exists Y_1, \ldots, Y_n \; \forall X_1, \ldots, X_n \; \triangleright (Y_1, X_1) \times \ldots \times \triangleright(Y_n, X_n) \models \mathcal{W}(C, \phi) \tag{3.21}$$

In this scenario we wonder if exists a *decomposition* of the formula $\mathcal{W}(C, \phi)$ as a *product formula* of the form:

$$\phi_1 \times \ldots \times \phi_n$$

where $\phi_1, \ldots, \phi_n$ are closed, unary formulas. Whenever such decomposition exists we require that each $\triangleright(Y_i, X_i)$ satisfies a formula $\phi_i$ of the product. We recall that for a product formula $\phi$, we write $\models \phi$ if $\phi$ is satisfied by all $n$-ary context system. In this case $\phi$ is *valid*. Moreover $\models_\times \phi$ if $\phi$ is satisfied by all $n$-product process $P_1 \times \ldots \times P_n$ of any context system. In this case $\phi$ is *weakly valid*.

According to [77] there is not a unique product formula $\phi_1 \times \ldots \times \phi_n$ such that $\models_\times \phi_1 \times \ldots \times \phi_n \Leftrightarrow \mathcal{W}(C, \phi)$.

Whenever $\phi$ is a *finite* formula, *i.e.*, it is neither LET MAX nor LET MIN, there exists a finite decomposition, *i.e.*, a finite collection of product formulae $\langle \phi_1^i \times \ldots \times \phi_n^i \rangle_{i \in I}$, where $I$ is and index set, such that

$$\models_\times \bigvee_{i \in I} \phi_1^i \times \ldots \times \phi_n^i \Leftrightarrow \mathcal{W}(C, \phi)$$

Without loss of generality, it is possible to consider that the decomposition is made in *dyadic* formula.

| | | |
|---|---|---|
| $(i) \models_\times \mathbf{T}$ | $\Leftrightarrow$ | $\mathbf{T} \times \mathbf{T}$ |
| $(ii) \models_\times \mathbf{F}$ | $\Leftrightarrow$ | $\mathbf{T} \times \mathbf{F} \vee \mathbf{F} \times \mathbf{F}$ |
| $(iii) \models_\times \phi_1 \times \psi_1 \wedge \phi_2 \times \psi_2$ | $\Leftrightarrow$ | $(\phi_1 \wedge \phi_2) \times (\psi_1 \wedge \psi_2)$ |
| $(iv) \models_\times \langle (a,b) \rangle (\phi \times \psi)$ | $\Leftrightarrow$ | $(\langle a \rangle \phi) \times (\langle b \rangle \psi)$ |
| $(v) \models_\times [(a,b)](\bigvee_i \phi^i \times \psi^i)$ | $\Leftrightarrow$ | $\bigvee_i ([a]\phi^i) \times ([b]\psi^i)$ |

where in $(v)$ $\bigvee_i \phi^i \times \psi^i$ is assumed to be saturated.

Table 3.1: Weak equivalences for decomposition of properties (see [77]).

**Definition 3.4** *A formula $\phi$ is said to be a* disjunctive product formula *if it has the form*

$$\phi = \bigvee_{i \in I} \psi^i \times \varphi^i$$

*for some finite collection $\langle \psi^i \rangle_{i \in I}$ and $\langle \varphi^i \rangle_{i \in I}$ of closed, monadic formulae,* i.e.*, there are not product of other formulae.*

**Definition 3.5** *A disjunctive product formula, $\bigvee_{i \in I} \psi^i \times \varphi^i$ is said to be* saturated*, provided for any product formula $\psi \times \varphi$*

$$\models \psi \times \varphi \Rightarrow \bigvee_{i \in I} \psi^i \times \varphi^i$$

*is equivalent to*

$$\models \psi \Rightarrow \psi^i \text{ and } \models \varphi \Rightarrow \varphi^i \text{ for some } i \in I$$

It is important to note that every general disjunctive product formula can be saturated (see [77]). In Table 3.1 there are recalled weak equivalences for decomposing properties.

The following theorem holds.

**Theorem 3.1 ([77])** *Let $\phi$ be a finite formula. Then there exists a weakly equivalent disjunctive product formula $\bigvee_{i \in I} \psi^i \times \varphi^i$ such that*

$$\models_\times \phi \Leftrightarrow \bigvee_{i \in I} \psi^i \times \varphi^i$$

Hence, whenever $\phi$ is a finite formula there exists a disjunctive product formula $\bigvee_{i \in I} \psi_1^i \times \ldots \times \psi_n^i$, that is weakly equivalent to $\phi$. In this way our problem becomes the following:

$$\exists Y_1, \ldots, Y_n \, \forall X_1, \ldots, X_n \, \triangleright (Y_1, X_1) \times \ldots \times \triangleright (Y_n, X_n) \models_\times \bigvee_{i \in I} \psi_1^i \times \ldots \times \psi_n^i \quad (3.22)$$

We have a disjunction of product formulas. In order to solve our problem it is sufficient to enforce one of the product formula fo the disjunction. Indeed we reduce the problem in Formula (3.22) as follows:

$$\exists Y_1, \ldots, Y_n \; \forall X_1, \ldots, X_n \; \triangleright (Y_1, X_1) \times \ldots \times \triangleright (Y_n, X_n) \models_{\times} \psi_1^k \times \ldots \times \psi_n^k \quad (3.23)$$

Since we can consider that we have a saturated formula, we can solve the problem above by solving $n$ problems like the following one:

$$\exists Y_j \; \forall X_j \; \triangleright (Y_j, X_j) \models \psi_j^k$$

## 3.3 Online path model checking

In this section we propose another technique to monitor security properties at run-time that is not based on the *open system* paradigm approach. As a matter of fact, starting from the idea of *model checking a path*, proposed by Markey and Schnoebelen in [83] for solving the model checking problem on a single path instead on the whole model, we present a technique for monitoring security properties.

Our goal is to control system at run-time. In particular we define controller operator able to control system and enforce several security properties. In this scenario we describe another approach based on the idea that only a trace of execution of the target $X$ is monitored. Hence it is possible to directly check the execution trace by using the partial model checking. We call this method *Online path model checking* ($OPMC$ for short).

Let $X$ be the target and $\phi$ the security property we would like to enforce. The main idea is that, if the formula obtained partially evaluating $\phi$ with respect to the action performed by $X$ is equivalent to false we halt the execution. This halt condition comes directly from the partial model checking function with respect to prefix operators (see Table 3.2). As a matter of fact, it means that the target is going to perform an action that violates the property.

### 3.3.1 Partial model checking for *Online Path Model Checking*

The scenario we consider is the same described at the beginning of the chapter. We have a system $S$ and a logic formula $\phi$ expressed in equational $\mu$-calculus. Let $X$ be one component that may be dynamically changed (*e.g.*, a downloaded mobile agent). We say that the system $S \| X$ enjoys $\phi$ if and only if for every behavior of the component $X$, the behavior of the system $S$ enjoys that security property, *i.e.*,

$$\forall X \quad (S \| X) \models \phi \quad (3.24)$$

First of all, by applying partial model checking, we reduce the problem in Formula (3.24) as follows:

$$\forall X \quad X \models \phi' \quad (3.25)$$

where $\phi' = \phi_{//_S}$. We assume to not know anything about $X$ a priori. We want to detect if $X$ is going to perform an action that violates $\phi$. On the contrary with respect to the previous approach (see Section 3.2) we do not check all the possible behavior of the target $X$ but we consider only the trace execution (*path*) we can observe at run-time. In order to do that we consider the following definition.

**Definition 3.6** *An execution $u = a_0 a_1 \ldots$ [4]is* secure *with respect to a formula $\phi$ whenever:*

$$\forall i \geq 0 \quad u^{[0\ldots i]} \models \phi$$

*where $u^{[0\ldots i]}$ is a simplification for $a_0 a_1 \ldots a_i$.*

*A formula $\phi$ is* initially satisfiable *if it is satisfiable in the initial state of the execution.*

$$u^{[0]} \models \phi$$

We describe an algorithm, called *Online path model checking*, that controls a target $X$, actually getting its actions one by one and analyzing them. The algorithm is said *online* because works without any initial knowledge on the state graph of the process (see [68]) and *path model checking* (see [83]) because we analyze the sequence of action of $X$.

Let $\phi$ be an equational $\mu$-calculus formula. It must be initially satisfiable. So at the beginning we check $\mathbf{0} \models \phi$. After every action $a$ performed by $X$, we apply the partial model checking for prefix operator (see Table 3.2) to the formula $\phi$ and we find $\phi_1 = \phi_{//_{a.t}}$. Then we require again that $\phi'$ is initially satisfiable. This algorithm ends if the execution of $X$ ends and in this case we discover that $X$ is a safe process or if there exists an action $a_i$ performed by $X$ such that $\mathbf{0} \not\models (\phi_{i-1})_{//_{a_i t}} = \phi_i$ where $\phi_{i-1}$ is the formula obtained by partial model checking with respect to the first $i-1$ actions performed by $X$. In this case the formula $\phi_i$ is equivalent to $\mathbf{F}$. Hence we conclude that the action $a_i$ is not legal.

In order to better explain these different methods we give a simple example.

**Example 3.7** *Let $\phi = [a][b]\mathbf{T}$ be a formula we want to satisfy. $\mathbf{0} \models \phi$ trivially.*

*We suppose that $X$ performs the action $a$. The monitor calculate the $\phi_1 = [b]\mathbf{T}$, according to Table 3.2 and checks if $\mathbf{0} \models \phi_1$. This holds trivially.*

*We suppose that $X = a.b.\mathbf{0}$.*

$$\xrightarrow{a} \cdot \xrightarrow{b} \mathbf{0}$$

*We have to check $X \models \phi$ but we do not know nothing about the behavior of $X$. We suppose $X$ performs the action $a$. We apply the partial model checking to $\phi$ with respect to the prefix operator of action $a$ and we check $\mathbf{0} \models \phi_{//_a}$. In this way, we shouldn't store what action was performed by $X$ because this information is store in the new formula $\phi_{//_a} = \phi_1$. If $\mathbf{0} \not\models \phi_1$ then we can deduce that the action $a$ is not compatible with the formula. On the other hand, if $\mathbf{0} \models \phi_1$ we look at the next action that $X$ could perform. If $X$ performs another action we apply again the same algorithm.*

[4]It is to note that, in order to describe the trace execution we adopt the same notation of linear temporal logic.

Renaming phase

$$
\begin{array}{lcl}
X//a.t_r & = & X_{t_r} \\
(X =_\sigma \phi D)//a.t_r & = & X_{t_r} =_\sigma \phi//a.t_r D//a.t_r \\
(\langle b\rangle\phi)//a.t_r & = & \langle b\rangle(\phi)//a.t_r \\
([b]\phi)//a.t_r & = & [b](\phi)//a.t_r \\
\phi_1 \wedge \phi_2//a.t_r & = & (\phi_1//a.t_r) \wedge (\phi_2//a.t_r) \\
\phi_1 \vee \phi_2//a.t_r & = & (\phi_1//a.t_r) \vee (\phi_2//a.t_r) \\
\mathbf{T}//a.t_r & = & \mathbf{T} \\
\mathbf{F}//a.t_r & = & \mathbf{F}
\end{array}
$$

Evaluating phase

$$
\begin{array}{lcl}
X//a.t & = & X_t \\
(X =_\mu \phi D)//a.t & = & \phi//a.tD//a.t[(X =_\mu \phi D)//a.t_r/X_{t_r}][\mathbf{F}/X_t] \\
(X =_\nu \phi D)//a.t & = & \phi//a.tD//a.t[(X =_\nu \phi D)//a.t_r/X_{t_r}][\mathbf{T}/X_t] \\
\langle b\rangle\phi//a.t & = & \begin{cases} (\phi//a.t_r) & \text{if } b = a \\ \mathbf{F} & \text{if } b \neq a \end{cases} \\
[b]\phi//a.t & = & \begin{cases} (\phi//a.t_r) & \text{if } b = a \\ \mathbf{T} & \text{if } b \neq a \end{cases} \\
\phi_1 \wedge \phi_2//a.t & = & (\phi_1//a.t) \wedge (\phi_2//a.t) \\
\phi_1 \vee \phi_2//a.t & = & (\phi_1//a.t) \vee (\phi_2//a.t) \\
\mathbf{T}//a.t & = & \mathbf{T} \\
\mathbf{F}//a.t & = & \mathbf{F}
\end{array}
$$

Table 3.2: Partial evaluation function for prefix operator.

This method has a lot of advantages. For instance, it is not necessary that all the sequence of actions are stored somewhere. Indeed the evaluation of the action is already stored in the new formula by partial model checking. Moreover, applying the partial evaluation function for prefix operator to the initial formula, it is able to understand immediately if this action compromise the security of the whole system or not.

It is possible to note that this technique gives the same results that we obtain by using the truncation operator $\triangleright_T$ for safety properties. As a matter of fact this technique permits to recognize a bad action but it does not give any advantage to repair it. Another advantage of this approach is that we are able to enforce all properties that can be expressed by an equational $\mu$-calculus formula.

### Computational analysis

Let $u$ be the sequence of actions performed by $X$. Let $n$ the length of $u$. Looking at the rule of partial model checking for prefix operator we have that the cost of each operation

of partial model checking on a formula $\phi$ is $\mathcal{O}(|\phi|)$, *i.e.*, it is linear in the size of the considered formula.

According to [34], we know that the cost of model checking of an equational $\mu$-calculus formula $\phi$ is $\mathcal{O}((s|\phi|)^{ad})$ where $s$ is the number of states of the model and $ad$ is the alternation depth of the formula $\phi$. In our method the model checking is always do with respect to the model $\mathbf{0}$ which number of state is $1$. So the cost of the whole algorithm is $n(\mathcal{O}(|\phi|^{ad}))$.

If we restrict to consider safety properties, the cost of the algorithm decreases. As a matter of fact, according to [27], safety properties can be expressed in $\mu$-calculus, by using only $\nu$ fixpoint, without diamond operator and the $\mu$-fixpoint. These kind of formulas are trivially alternation-free. Referring to [34, 46], we know that, for this class of formulae it is possible to solve the model checking problem in $\mathcal{O}(|\phi|s)$ where $s$ is the number of the state of the transition system and $|\phi|$ is the size of the formula.

In this case the cost of the whole algorithm is $(n)\mathcal{O}(|\phi|)$ where $n$ is the length of the sequence of actions performed by $X$.

## 3.4 Related work on enforcement mechanisms

Security automata was introduced by Schneider in [124]. A security property that can be enforced in this way corresponds to a safety property.

Starting from the Schneider's work, Ligatti *et al.* in [19, 20] have defined four different kinds of security automata which deal with finite sequences of actions: *truncation automaton, suppression automaton, insertion automaton* and *edit automaton*.

In Section 3.2.1 we have showed a possible way to model security automata in [19, 20] by using process algebra. Moreover we study also systems in a timed setting.

Other works present different frameworks to model, analyze and study security automata. For instance, in [96], a process algebra is used to model part of the usage control framework (see [108]) for the $JVM^{\text{tm}}$. As a matter of fact, security in the grid environment is a challenging issue. The authors proposed to integrate a local monitor into the grid computational service architecture, to control the behavior of applications executed on behalf of grid users. Their approach was inspired to the concept of *continuous usage control*.

More recently, [17] proposes to use $CSP - OZ$, a specification language combining *Communicating Sequential Processes* ($CSP$) and *Object-Z* ($OZ$), to specify security automata, formalize their combination with target systems, and analyze the security of the resulting system specifications. They provide theoretical results relating $CSP - OZ$ specifications and security automata and show how refinements can be used to reason about specifications of security automata and their combination with target systems. We also use process algebra to model security automata and describe their combination with a target system.

Also Bartoletti, Degano and Ferrari in [13] refer to [124] by saying that while safety properties can be enforced by an execution monitor, liveness properties cannot. In order to

enforce safety and liveness properties, they enclose security-critical code in *policy framings*, in particular *safety framings* and *liveness framings*, that enforce respectively safety and liveness properties of execution histories. This is however a static analysis that over-approximates behavior *history expressions*. In [12] they have proposed a mixed approach to access control, that efficiently combines static analysis and run-time checking. They compile a program with policy framings into an equivalent one without framings, but instrumented with local checks. The static analysis determines which checks are needed and where they must be inserted to obtain a program respecting the given security requirements. The execution monitor is essentially a finite-state automaton associated with the relevant security policies. In our work we isolate the possible un-trusted components by partial model checking then we checks at run-time the target. An advantages of our method with respect to the one described in [12] is that our monitors not only halts the target whenever it are going to violate the given policy, but only they provide some mechanisms to enforce the policy without ending the target execution by the insertion or the suppression of some possible *wrong* actions. Moreover, in Section 3.2.2 we have also defined several semantics definitions of controller operators able to enforce also information flow properties. In [87], a preliminary work has been provided that is based on different techniques for automatically synthesizing systems enjoying a very strong security property, *i.e.*, $SBSNNI$ (see [53]). That work did not deal with controllers.

Much of prior work are about the study of enforceable properties and related mechanisms. In [44] authors deal with a safety interface that permits to study if a module is safe or not in a given environment. Here is checked all system, instead in our approach, through the partial model checking function, we are able to monitor only the necessary/untrusted part of the system.

**Related work on enforcement mechanisms for distributed systems**

In Section 3.2.4 we deal with system in which there are several unspecified components by applying the theory of compositionality by contexts proposed in [77]. In particular we show different control approaches.

Other works deal with partially specified system using context. In [64] is presented a framework inspired by [77] for the validation of reactive system embedded in a test environment, or isolated from their operational environment, thereby inducing a natural classification of validation strategies in different scenario. However they do not deal with security property and do not make security analysis. Moreover the authors consider context with only one unknown component, instead in our work we present results on contexts in which more than one component of the system is unspecified.

In [106], contexts are used to build *upgrade specification* from components and their interface languages. No security problem is addressed.

A lot of works deal with the problem of decentralized discrete-event control problems, as [10, 11, 32, 114, 120] the authors have studied the decentralized supervisory control problem of discrete event systems under partial observation. They do not treat this problem from a security point of view. As a matter of fact they do not do security analysis by

considering generating controller for whatever possible behavior of the unspecified part, *i.e.*, they do not consider the unspecified part of the system as a potential attacker. In [32, 114, 120] the authors have investigated on the necessary and sufficient conditions for the existence of decentralized supervisors for ensuring that the controlled behavior of the system lies in a given range.

**Related work on model checking a path**

A great deal of work is done for developing techniques to cope with the run-time model checking. A technique that is used in run-time verification is the *model checking a path*, *i.e.*, solving the model checking problem on a single path instead on the whole model. It was introduced by Markey and Schnoebelen in [83].

In particular, this technique is developed for model checking of linear time logic. Using *standard dynamic programming methods* a path can obviously be checked in bilinear time, $O(|path| \times |formula|)$. In [84], they prove that model checking a path of modal $\mu$-calculus formulae has the complexity of $O((n \times |\phi|)^{ad})$ We can note that this technique is developed on $LTL$ formulas and, in [83], the authors give the cost of the algorithm for $LTL$ formulas and for $LTL + Past$ formulae. These two logics are suitable for express safety properties. However, we use the equational $\mu$-calculus that it is more expressive that $LTL$ and $PLTL$.

Previous works have deal with the state explosion in model checking problem. We focus in particular on two techniques, *on-the-fly model checking* and *online model checking*. These two techniques do not work on a path or a trace of execution but consider at run-time a state graph. The main difference between these two kind of model checking is that in the first one the state graph of the process is known but it is not stored in memory. Instead in the second one nothing is known about the program and the state graph in built and check at run-time. Both of this model checking are developed by Jard and Jeron: Online model checking is described for the first time in [68] and the on the fly model checking is developed in [69]. In [68] Jard e Jeron introduce the *online model checking* where satisfiability is checked during the state generation process. In particular they work with $TL$ formulas. The basic idea of their paper is to check during the state enumeration. For that aim, the temporal logic specification must be executed. It is done by first translating the logic specification into a finite automaton. Here is the main algorithmic difficulty. The automaton will value the system states during enumeration. The decision of validity or rejection can be reached in finite time providing a large enough memory to store a number of state equal to the depth of the state graph. This approach is different from the approach of *on-the-fly model checking* that was introduced by Jard, Jeron *et al.* in the technical report [69]. This method is used at runtime. The global state graph of the automata is not stored before the labeling but there are generate only the part of the state graph that are interested at runtime that are necessary to verify the satisfiability of the formula. This second kind of model checking is more used than the other.

# Chapter 4

# Synthesis of controller programs

In this chapter we present our mechanisms for the synthesis of controller programs (see Chapter 3).

The *synthesis problem* occurs when we deal with a system in which there are some unspecified components, *e.g.*, a software that is not completely implemented.

Following the approach based on the open system paradigm for the specification and verification of secure systems, in Chapter 3 we have defined several process algebra controller operators, developed for enforcing security properties by using controller programs that monitor the possible un-trusted component in order to guarantee the whole system is secure.

In this chapter we deal with the problem of the synthesis of controller programs. Indeed, given a system and a property, we wonder if there exists an implementation of a controller program that, by controlling the unspecified components, makes the system secure, *i.e.*, it guarantees the system satisfy the given property whatever the behavior of the unspecified components that it controls is. In particular given a system, that we want to be secure, a security property we want to enforce and a controller operator we are going to use in order to do that, we show how and when it is possible to synthesize a controller program for the given controller operator.

The technical proofs of the results in this chapter are in the Appendix A.2.

## 4.1 Synthesis of controller programs for safety properties

In Section 3.2.1, we have described four different controller operators: $Y \triangleright_T X$, $Y \triangleright_S X$, $Y \triangleright_I X$ and $Y \triangleright_E X$ whose semantics definitions are recalled in Table 4.1. The problem we want to solve is the following:

$$\exists Y \quad \forall X \quad S \| Y \triangleright_{\mathbf{K}} X \models \phi$$

where $\phi$ is a security property and $\mathbf{K} \in \{T, S, I, E\}$.

By applying partial model checking, we evaluate the behavior of $S$ into the formula

Truncation:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd_T F \xrightarrow{a} E' \rhd_T F'}$$

Suppression:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd_S F \xrightarrow{a} E' \rhd_S F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \rhd_S F \xrightarrow{\tau} E' \rhd_S F'}$$

Insertion:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \rhd_I F \xrightarrow{a} E' \rhd_I F'} \qquad \frac{E \not\xrightarrow{a} E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \rhd_I F \xrightarrow{b} E' \rhd_I F}$$

Edit:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{a} E' \rhd_E F'} \qquad \frac{E \xrightarrow{-a} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{\tau} E' \rhd_E F'}$$

$$\frac{E \not\xrightarrow{a} E' \quad E \xrightarrow{+a.b} E' \quad F \xrightarrow{a} F'}{E \rhd_E F \xrightarrow{b} E' \rhd_E F}$$

Table 4.1: Semantics definition of controller operators for enforcing safety properties.

$\phi$. Hence we reduce the previous problem in the following one.

$$\exists Y \quad \forall X \quad Y \rhd_{\mathbf{K}} X \models \phi' \tag{4.1}$$

where $\phi' = \phi_{//S}$.

The equation in Formula (4.1) might not be easy to manage because of the presence of the universal quantification on all possible behaviors of the target $X$. For that reason, firstly, we underline that, by $\rhd_{\mathbf{K}}$ operators, we are goring to enforce safety properties, as we have already said in Section 3.2.1. Hence we restrict ourselves to consider a subclass of equational $\mu$-calculus formulas that we call $Fr_\mu$. It consists of equational $\mu$-calculus formulas without $\langle\_\rangle$ operator.

It is easy to prove that, according to the rule of the partial evaluation function with respect to parallel operator, this set of formulas is closed under the partial model checking function and that the following result holds.

**Proposition 4.1 ([29])** *Let $E$ and $F$ be two processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$. The same result holds also if $F$ and $E$ are strong similar.*

Now, let us consider again the problem in Formula (4.1). Referring to the previous proposition, it is possible to note that whenever:

**Assumption 4.1** *For every $X$ and $Y$, we have:*

$$Y \rhd_{\mathbf{K}} X \preceq Y$$

then the the problem in Formula (4.1) can be equivalently reduced as follows:

$$\exists Y \quad Y \models \phi' \tag{4.2}$$

The formulation (4.2) is easier to be managed.

According to the semantics definition of $\triangleright_{\mathbf{K}}$ operators given in Section 3.2.1 and recalled in Table 4.1, we are able to give the following proposition.

**Proposition 4.2** *For every* $\mathbf{K} \in \{T, S, I, E\}$

$$Y \triangleright_{\mathbf{K}} X \preceq Y[f_{\mathbf{K}}]$$

*holds, where* $f_{\mathbf{K}}$ *is a relabeling function depending on* $\mathbf{K}$*. In particular,* $f_T$ *is the identity function on* $Act$[1] *and*

$$f_S(a) = \begin{cases} \tau & \textit{if } a = -a \\ a & \textit{othw} \end{cases} \qquad f_I(a) = \begin{cases} \tau & \textit{if } a = +a \\ a & \textit{othw} \end{cases}$$

$$f_E(a) = \begin{cases} \tau & \textit{if } a \in \{+a, -a\} \\ a & \textit{othw} \end{cases}$$

This proposition provides that, whatever controller operators $\triangleright_{\mathbf{K}}$ is chosen to use to enforce a given safety properties, it is possible to find a solution for the problem in Formula 4.1 by finding a controller program $Y$ such that:

$$Y[f_{\mathbf{K}}] \models \phi'$$

To further reduce the previous formula, we can use the partial model checking function for relabeling operator, calculating $\phi''_{\mathbf{K}} = \phi'_{//_{[f_{\mathbf{K}}]}}$ for each $\mathbf{K}$. Thus we obtain:

$$\exists Y \quad Y \models \phi''_{\mathbf{K}} \tag{4.3}$$

This is a satisfiability problem in $\mu$-calculus. Hence a possible model $Y$ for $\phi''_{\mathbf{K}}$ can be find according to the Theorem 2.2. So we are able to prove the following result

**Theorem 4.1** *The problem described in Formula (4.1) is decidable.*

Note that the trivial solution exists. As a matter of fact the process $\mathbf{0}$ is a model for all possible formulas in $Fr_\mu$. As a matter of fact for every process $P$, $0 \preceq P$, hence, according to the Proposition 4.1, $0 \models \phi$. Referring to the semantics definition of each controller operator, for each $\mathbf{K}$ and

$$\forall\, X, \quad 0 \triangleright_{\mathbf{K}} X = 0$$

Hence

$$\text{for each } \mathbf{K},\ \forall\, X \quad 0 \triangleright_{\mathbf{K}} X \models \phi'$$

[1] Here the set $Act$ must be consider enriched by control actions.

### 4.1.1 Automated synthesis of maximal controller program for truncation operator

In this section we recall the notion of *maximal model* with respect to the simulation relation and show how it is possible to synthesize a *maximal program controller* $Y$ for the operator $Y \triangleright_T X$.

Firstly, we give the definition of *maximal model with respect to the simulation relation* as follow (see [117, 118]).

**Definition 4.1** *A process $E$ is said to be the* maximal model for a given formula $\phi$ *with respect to the relation of simulation if and only if $E \models \phi$ and $\forall E'$ such that $E' \models \phi$ we have that $E' \preceq E$.*

Informally, the maximal program controller $Y$ is the process that restricts as little as possible the activity of the target $X$.

In order to find the maximal model we exploit the theory developed by Walukiewicz in [131].

Usually the discovered model is a non-deterministic process. There are several possible solution to find deterministic model. For instance, it is possible to turn $E$ into a deterministic model but some deadlock may occur.

Let us consider a subset of formulas of $Fr_\mu$ without $\vee$. This set of formulas is called the *universal conjunctive $\mu$-calculus formulas, $\forall_\wedge \mu C$.*

**Definition 4.2** *The set $\forall_\wedge \mu C$ of* universal conjunctive $\mu$-calculus formulas *is the largest subset of equational $\mu$-calculus formulas that can be written without either the $\vee$ operator and the $\langle \_ \rangle$ modality.*

It is possible to prove that $\forall_\wedge \mu C$ is closed under the partial model checking function.

**Proposition 4.3 ([59])** *$\forall_\wedge \mu C$ is closed under the partial model checking function.*

Moreover, for this class of formulas the following result holds.

**Proposition 4.4 ([58])** *Given a formula $\phi \in \forall_\wedge \mu C$, a model $E$ of this formula exists.*

In order to generate the maximal model $E$, we find a model for $\phi \wedge \psi$ where $\psi = X$, $X =_\nu \bigwedge_{a \in Act \setminus \{\tau\}} ([a]\mathbf{F} \vee (\langle a \rangle X \wedge [a]X))$. The formula $\psi$ permits us to check all possible actions in $Act$.

First of all we prove that the formula $\phi \wedge \psi$ is satisfiable.

**Lemma 4.1** *Let $\phi \in \forall_\wedge \mu C$ and $\psi = X$ where $X =_\nu \bigwedge_{a \in Act} ([a]\mathbf{F} \vee (\langle a \rangle X \wedge [a]X))$. If $\phi$ is satisfiable then $\phi \wedge \psi$ is satisfiable.*

Exploiting the theory of Walukiewicz, we find a deterministic model $E$ for $\phi \wedge \psi$ that does not perform $\tau$ actions. As a matter of fact, we assume that the pre-model (and consequently the canonical structure) is built using quasi-models where the $(or_{left})$ is applied only if the $(or_{right})$ fails. With this assumption, since we will apply the canonical

model only to one kind of formula with disjunction, we may control which branch will be followed and so the kind of canonical model generated. In this way we generate a deterministic model of $\phi$. We prove the following proposition.

**Proposition 4.5** *Given a formula $\phi \in \forall_{\wedge}\mu C$, a maximal deterministic model $E$ of this formula exists.*

In particular, as model we consider the canonical structure (see [131]) of the formula $\phi \wedge \psi$. It is easy to prove the following lemma.

**Lemma 4.2** *Let $E' \models \phi$ with $\phi \in \forall_{\wedge}\mu C$. Let $E$ be the canonical structure of $\phi \wedge \psi$, then the following relation holds:*

$$E' \preceq E$$

Hence $E$ is the maximal model for $\phi$.

Thus, by using the result in Proposition 4.5, it is possible to find a maximal deterministic model that synthesizes a controller operator to force a security policy, *i.e.*, the synthesis of a truncation automaton for a component that will allow the whole system to enjoy the desired security property.

Note that the maximal model that we have found it is not necessary unique because of the simulation relation is a pre-order.

### 4.1.2 Timed setting

In this section we present how the results that we have proved in the previous section, can be obtained also in a timed setting. In particular we are able to synthesize controller programs for the controller operators defined in Table 4.1 also for systems that work in a timed setting.

Referring to Section 2.3.2, we use the $tick$ action to model the elapsing of time. The introduction of the $tick$ action in the set of possible action $Act$ does not change the semantics definition of the controller operators $\triangleright_{\mathbf{K}}$, where $\mathbf{K} \in \{T, S, I, E\}$.

We prove the following proposition in order to show that $\triangleright_{\mathbf{K}}$ operators permit the elapsing of time, *i.e.*, are weakly time alive (see Section 2.3.3).

Let $E$ and $F$ be finite state processes, in particular $E$ is the controller program and $F$ the target. The following proposition holds.

**Proposition 4.6** *If both $E$ and $F$ are weakly time alive, also $E \triangleright_{\mathbf{K}} F$ is weakly time alive.*

According to Section 3.2.1, the $tick$ action is performed when both the target and the controller perform it. Also in this case we can followed the same reasoning made before.We restrict ourselves to consider formulas in $Fr_{\mu}$, the following proposition holds.

**Proposition 4.7** *Let $E$ and $F$ be processes and $\phi \in Fr_{\mu}$. If $F \preceq_t E$ then $E \models \phi \Rightarrow F \models \phi$.*

Thus also in a timed setting we require that the following assumption is verified.

**Assumption 4.2** *For every $X$ and $Y$, we have:*

$$Y \triangleright_{\mathbf{K}} X \preceq_t Y$$

If $X$ and $Y$ satisfy the Assumption 4.2 then the property in Formula $(4.1)$ is equivalent to:

$$\exists Y \quad Y \models \phi' \tag{4.4}$$

The Formulation (4.4) is easier to be managed. A proposition similar to Proposition 4.2 holds. In particular, looking at the definition of weak timed simulation and at the proof of the Proposition 4.2, it is possible to prove the following proposition.

**Proposition 4.8** *For every $\mathbf{K} \in \{T, S, I, E\}$ $Y \triangleright_{\mathbf{K}} X \preceq_t Y[f_{\mathbf{K}}]$ holds, where $f_{\mathbf{K}}$ is a relabeling function depending on $\mathbf{K}$. In particular, $f_T$ is the identity function on $Act$[2] and*

$$f_S(\alpha) = \begin{cases} \tau & \text{if } \alpha = -\alpha \\ \alpha & \text{othw} \end{cases} \qquad f_I(\alpha) = \begin{cases} \tau & \text{if } \alpha = +\alpha \\ \alpha & \text{othw} \end{cases}$$

$$f_E(\alpha) = \begin{cases} \tau & \text{if } a \in \{+\alpha, -\alpha\} \\ \alpha & \text{othw} \end{cases}$$

At this point in order to satisfy the Formula (3.25) it is sufficient to find a controller program $Y$ s.t.:

$$Y[f_{\mathbf{K}}] \models \phi'$$

To further reduce the previous formula, we can use the partial model checking function for relabeling operator, calculating $\phi''_{\mathbf{K}} = \phi'_{//_{[f_{\mathbf{K}}]}}$ for each $\mathbf{K}$. Thus we obtain:

$$\exists Y \quad Y \models \phi''_{\mathbf{K}} \tag{4.5}$$

This is a satisfiability problem in $\mu$-calculus that can be solved by Theorem 2.2. So we are able to prove the following result

**Theorem 4.2** *The problem described in Formula (4.1) is decidable also in a timed setting.*

### 4.1.3 An example

Let us consider the process $S = a.b.\mathbf{0}$ and the following equational definition $\phi = Z$ where $Z =_\nu [\tau]Z \wedge [a]W$ and $W =_\nu [\tau]W \wedge [c]\mathbf{F}$. It asserts that, after every action $a$, an action $c$ cannot be performed. Let $Act = \{a, b, c, \tau, \bar{a}, \bar{b}, \bar{c}\}$ be the set of actions. Applying the partial evaluation for the parallel operator we obtain, after some simplifications, the following system of equation, that we denoted with $\mathcal{D}$.

$$\begin{array}{l} Z_{//_S} =_\nu [\tau]Z_{//_S} \wedge [\bar{a}]Z_{//_{S'}} \wedge [a]W_{//_S} \wedge W_{//_{S'}} \\ W_{//_{S'}} =_\nu [\tau]W_{//_{S'}} \wedge [\bar{b}]W_{//_{\mathbf{0}}} \wedge [c]\mathbf{F} \\ Z_{//_{S'}} =_\nu [\tau]Z_{//_{S'}} \wedge [\bar{b}]Z_{//_{\mathbf{0}}} \wedge [a]W_{//_{S'}} \\ W_{//_S} =_\nu [\tau]W_{//_S} \wedge [\bar{a}]W_{//_{S'}} \wedge [c]\mathbf{F} \\ Z_{//_{\mathbf{0}}} = \mathbf{T} \\ W_{//_{\mathbf{0}}} = \mathbf{T} \end{array}$$

[2]Here the set $Act$ must be consider enriched by control actions.

where $S \xrightarrow{a} S'$ so $S'$ is $b.\mathbf{0}$.

The information obtained through partial model checking can be used to enforce a security policy. In particular, choosing one of the four operators and using its definition we simply need to find a process $Y[f_{\mathbf{K}}]$, where $\mathbf{K}$ depend on the chosen controller, that is a model for the previous formula.

In this simple example we choose the controller operator $\triangleright_S$. Hence we apply the partial model checking for relabeling function $f_S$ to the previous formula, that we have simplified replacing $W_{//_\mathbf{0}}$ and $Z_{//_\mathbf{0}}$ by $\mathbf{T}$ (and assumed that $Y$ can only suppress $c$ actions). We obtain $\mathcal{D}_{//_{f_S}}$ as follows.

$$
\begin{array}{l}
Z_{//_{S,f_S}} =_\nu [\tau]Z_{//_{S,f_S}} \wedge [-c]Z_{//_{S,f_S}} \wedge [\bar{a}]Z_{//_{S',f_S}} \wedge [a]W_{//_{S,f_S}} \wedge W_{//_{S',f_S}} \\
W_{//_{S',f_S}} =_\nu [\tau]W_{//_{S',f_S}} \wedge [-c]W_{//_{S',f_S}} \wedge [\bar{b}]\mathbf{T} \wedge [c]\mathbf{F} \\
Z_{//_{S',f_S}} =_\nu [\tau]Z_{//_{S',f_S}} \wedge [-c]Z_{//_{S',f_S}} \wedge [\bar{b}]\mathbf{T} \wedge [a]W_{//_{S',f_S}} \\
W_{//_{S,f_S}} =_\nu [\tau]W_{//_{S,f_S}} \wedge [-c]W_{//_{S,f_S}} \wedge [\bar{a}]W_{//_{S',f_S}} \wedge [c]\mathbf{F}
\end{array}
$$

We can note the process $Y = a. - c.\mathbf{0}$ is a model of $\mathcal{D}_{//_{f_S}}$. Then, for any component $X$, we have $S\|(Y \triangleright_S X)$ satisfies $\phi$. For instance, consider $X = a.c.\mathbf{0}$. Looking at the first rule of $\triangleright_S$, we have:

$$(S\|(Y \triangleright_S X)) = (a.b.\mathbf{0}\|(a. - c.\mathbf{0} \triangleright_S a.c.\mathbf{0})) \xrightarrow{a} (a.b.\mathbf{0}\|(-c.\mathbf{0} \triangleright_S c.\mathbf{0}))$$

Using the second rule we eventually get:

$$(a.b.\mathbf{0}\|(-c.\mathbf{0} \triangleright_S c.\mathbf{0})) \xrightarrow{\tau} (a.b.\mathbf{0}\|\mathbf{0} \triangleright_S \mathbf{0})$$

and so the system still preserves its security since the actions performed by the component $X$ have been prevented from being visible outside.

## 4.2 Synthesis of controller programs for $BNDC$

In Section 3.2.2 we have presented several definitions of controller operators able to enforce information flow properties defined as $BNDC$ property. In Table 4.2 we remaind the semantics definition of these controllers.

As we have said in Section 3.2.2 the problem we want to solve is the following:

$$\exists Y \quad \forall X \quad S\|(Y \triangleright^* X)\backslash H \models \phi \tag{4.6}$$

where $\triangleright^*$ denoted a general controller operators.

In this section we provide a framework for synthesizing controller program $Y$ for a given controller operators able to enforce $BNDC$ properties.

As we have already said, by using the partial model checking approach we focus on the properties that the controller operator $Y$ has to enforce. Hence the problem in Formula (4.6) becomes:

$$\exists Y \quad \forall X \quad Y \triangleright^* X \models \phi_{S,\backslash H} \tag{4.7}$$

$\triangleright'$ operator:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F'} \qquad \frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F}$$

$\triangleright''$ operator

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F'} \qquad \frac{E \xrightarrow{a} E' \qquad F \overset{a}{\not\rightarrow}}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F}$$

Table 4.2: Semantics definition of controller operators for enforcing information flow properties.

We make the following assumption to deal with the universal quantification on all possible behavior of the target $X$. Then we will prove that the operators we have defined satisfy this assumption.

**Assumption 4.3** *For every $X$ and $Y$, we have:*

$$Y \triangleright^* X \sim Y$$

We are able to prove that the operator $\triangleright'$ and the operator $\triangleright''$ enjoy Assumption 4.3.

**Proposition 4.9** *The operators $\triangleright'$ and $\triangleright''$ enjoy Assumption 4.3.*

Note that for $BNDC$ properties it is sufficient that $Y \triangleright^* X$ and $Y$ are weakly bisimilar. Since every strong simulation is also a weak one [103] then $Y \triangleright^* X \approx X$. Hence, according to Proposition 4.9, both these operators could be applied to enforce information flow properties.

While designing such a process $Y$ could not be difficult in principle, we can take advantage of our logical approach and obtain an automated procedure as follows. As a matter of fact, the property expressed by the Formula $(4.7)$ is equivalent to:

$$\exists Y \quad Y \models \phi' \tag{4.8}$$

This is a satisfiability problem in $\mu$-calculus that can be solved by Theorem 2.2.

So we are able to prove the following result

**Theorem 4.3** *The problem described in Formula (4.7) is decidable.*

### Feasibility issues for our controllers

The introduction of a controller operator helps to guarantee a correct behavior of the entire system.

We discuss in this section the feasibility of our controller operators, *i.e.*, how and also if, the controllers $\triangleright'$, $\triangleright''$, can be implemented.

For the first controller operator, $\triangleright'$, we can note that it may in any moment neglect the external agent $X$ behavior, unless $X$ performs $\tau$. The behavior of the system may simply follow the behavior of the controller process. In particular, the controller may always choose to perform its correct action, rather than waiting for an action by the target. Thus, it would be easily implementable.

The operator $\triangleright''$ cannot be implemented if we are not able to decide a priori which are possible next steps that the external agent is going to perform. On the contrary, if it is possible to know a priori which is the set of possible next steps the target is going to perform, it would be possible to give priority to the first rule allowing always the correct action of the target. Thus, controller $\triangleright''$ leaves that the external agent executes correct action, if the first rule can be applied, and denies the unwanted situation checking them by the second rule. Also in this case internal actions performed by $X$ are permitted without any action of $Y$.

### 4.2.1 Timed setting

As we have showed in Section 3.2.2, it is possible to define process algebra controller operators to enforce information flow also for system that works in a timed settings. In Table 4.3 we recall the semantics definitions of such operators. As we have said in Section

---

$\triangleright'$ operator:

$$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\alpha} F'}{E \triangleright' F \xrightarrow{\alpha} E' \triangleright' F'} \quad \alpha \neq \tau \qquad \frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F} \qquad \frac{F \xrightarrow{\tau} F'}{E \triangleright' F \xrightarrow{\tau} E \triangleright' F'}$$

$\triangleright''$ operator:

$$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\alpha} F'}{E \triangleright'' F \xrightarrow{\alpha} E' \triangleright'' F'} \quad \alpha \neq \tau \qquad \frac{E \xrightarrow{a} E' \quad F \overset{a}{\not\Rightarrow} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F} \qquad \frac{F \xrightarrow{\tau} F'}{E \triangleright'' F \xrightarrow{\tau} E \triangleright'' F'}$$

---

Table 4.3: Semantics definition of controller operators for enforcing $tBNDC$ properties.

3.2.2 we model information flow properties in a timed settings by $tBNDC$ properties and we would like to solve the following problem:

$$\exists Y \quad Y \triangleright^* X \models \phi' \tag{4.9}$$

where $\phi'$ is the formula that expresses the $tBNDC$ properties after the partial evaluation w.r.t. the known part of the system. The controller operators has to satisfy the following assumption.

**Assumption 4.4** *For every $X$ and $Y$, we have:*

$$Y \triangleright^* X \approx_t Y$$

If the controller operator satisfies the Assumption 4.4, the Formula $(4.9)$ is equivalent to:

$$\exists Y \quad Y \models \phi' \tag{4.10}$$

As a matter of fact, the previous assumption permits us to conclude that $Y \triangleright^* X$ and $Y$ are timed bisimulation equivalent. It is possible to reduce the Formula $(4.9)$ to the Formula $(4.10)$ by resorting to the concept of characteristic formula for timed equivalence (Definition 2.29).

It is important to note that the Assumption 4.4 is a sufficient condition to enforce $tBNDC$. We have also to require not only that $Y$ satisfies the formula $\phi'$ but also that is weakly time alive because it has to permit the elapsing of time. As a matter of fact, since the $tick$ action can be performed by the system if both $Y$ and $X$ agree to perform it, if the $Y$ does not permit the elapsing of time this could generate a flow of information. The weakly time alive property for a finite state process $P$ can be expressed by a $\mu$-calculus formula as follows:

$$\phi_{w.t.a.} = Z =_\nu \langle\langle tick \rangle\rangle \mathbf{T} \wedge [\_]Z$$

Hence we have to find a model for $\phi' \wedge \phi_{w.t.a.} = \phi''$. Thus we obtain:

$$\exists Y \quad Y \models \phi'' \tag{4.11}$$

While designing such a process $Y$ could not be difficult in principle, we can take advantage of our logical approach and obtain an automated procedure. As matter of facts, exploiting the Theorem 2.2, it is possible to decide if there exists a model $Y$ for $\phi''$ and find it. For the semantics of conjunction, if $Y$ satisfies $\phi''$ it satisfies $\phi$. Hence $Y$ is suitable for Formula (4.11).

**Proposition 4.10** *Let $E$ and $F$ be two finite-state processes. If both $E$ and $F$ are weakly time alive, also $E \triangleright' F$ and $E \triangleright'' F$ are weakly time alive.*

**Proposition 4.11** *The operator $\triangleright'$ and $\triangleright''$ enjoy Assumption 4.4.*

So we are able to prove the following result.

**Theorem 4.4** *The problem described in Formula (4.7) is decidable for finite-state processes in a timed setting.*

### 4.2.2 An example

Let us consider the process $S = l.\mathbf{0} + h.h.l.\mathbf{0}$. The system $S$ where no high level activity is present is timed weakly bisimilar to $l.\mathbf{0}$. Let us consider the following equational definition:

$$Z_S =_\nu ([\tau]Z_S) \wedge [l]\mathbf{T} \wedge \langle\langle l \rangle\rangle \mathbf{T}$$

It asserts that a process may and must perform the visible action $l$. As for the study of $tBNDC$-like properties we can apply the partial evaluation for the parallel operator we obtain after some simplifications:

$$(Z_S)_{//S} \quad =_\nu \quad ([\tau](Z_S)_{//S}) \wedge [\overline{h}]\langle\langle\overline{h}\rangle\rangle\mathbf{T}$$

which, roughly, expresses that after performing a visible $\overline{h}$ action, the system reaches a configuration such that it must perform another visible $\overline{h}$ action. The information obtained through partial model checking can be used to enforce a security policy which prevents a system from having certain information leaks. In particular, if we use the definition of the controller as $\triangleright''$, we simply need to find a process that is a model for the previous formula, say $Y = \overline{h}.\overline{h}.\mathbf{0}$. Then, for any component $X$, we have $(S\|(Y \triangleright'' X)) \setminus \{h\}$ satisfies $(Z_S)_{//S}$.

For instance, consider $X = \overline{h}.\mathbf{0}$, we obtain:

$$(S\|(Y \triangleright'' X)) \setminus \{h\} \xrightarrow{\tau} (h.l.\mathbf{0}\|(\overline{h} \triangleright'' \mathbf{0})) \setminus \{h\}$$

Thus, using the second rule the controller may force to issue another $\overline{h}$ and thus we eventually get:

$$(h.l.\mathbf{0}\|(\overline{h} \triangleright'' \mathbf{0})) \setminus \{h\} \xrightarrow{\tau} (l.\mathbf{0}\|(\mathbf{0} \triangleright'' \mathbf{0})) \setminus \{h\} \approx l.\mathbf{0}$$

and so the system still preserves its security since the actions performed by the component $X$ have been prevented from being visible outside. On the contrary, if the controller would not be there would be a deadlock after the first internal action.

## 4.3 Synthesis of controller programs for composition of properties

In this section we show how our technique can be used also for synthesizing controller programs able to enforce *composition of properties*. In particular we wonder if there exists a way easier than the method described before, to enforce a property described by a formula $\phi$ that can be written as conjunction of sub-formulas $\phi_i$ *simpler* than itself, *i.e.*, the size of each $\phi_i$ is minor of the size of $\phi$. Thus, we present a method to enforce this kind of properties by using the truncation operator (see Section 3.2.1). In particular we prove that the composition of controller programs for truncation operator enforce the composition of properties they enforce. It is important to note that, since we consider the truncation operator, we are working under the additional assumption that the the properties that we investigate are safety properties, *i.e.*, we consider formulas in $Fr_\mu$ (see Section 4.1).

Hence we would like that the following relation holds:

$$\forall X \quad S\|X \models \phi \equiv \phi_1 \wedge \ldots \wedge \phi_n \tag{4.12}$$

where $\phi_1, \ldots, \phi_n$ are safety properties simpler than $\phi$. In order to guarantee that the whole system satisfy $\phi$ we have to find a controller program $Y$ that forces $\phi$ to be satisfied *i.e.*, :

$$\exists Y \quad \forall X \quad S\|Y \triangleright_T X \models \phi_1 \wedge \ldots \wedge \phi_n \tag{4.13}$$

According to Theorem 2.2, the cost of the satisfiability procedure is exponential in the size of the formula.

Here we present a method to find a controller program $Y$ for $\phi$ starting from controller operators of its sub-formulas $\phi_i$. As a matter of fact, let $\phi = \bigwedge_{i=1}^{n} \phi_i$ be the given formula, then by exploiting he Theorem 2.2, we synthesize a controller program $Y_i$ for each of $\phi_i$ formula. Finally, by composing $Y_i$ one to each other we obtain $Y$. This method is less expensive than synthesize directly $Y$. As a matter of fact, according to the Theorem 2.2, it is possible to find a model for a $\mu$-calculus formula $\phi$ has a cost exponential in the size of $\phi$, *i.e.*, let us consider that all the $\phi_i$ have the same size $m$, then the size of $\phi$ is $m \times n$. Hence synthesize directly $Y$ costs $\mathcal{O}(c^{m \times n})$.

On the other hand, the cost of our method is $n\mathcal{O}(c^m)$ because the cost for synthesizing $n$ models, one for each formula $\phi_i$, is $n\mathcal{O}(c^m)$ and the cost of the composition through the $\triangleright_T$ operator is constant in the size of the formula.

In order to describe our method, first of all, we rewrite Formula (4.12), by exploiting the semantics definition of the logical conjunction, as follows:

$$\begin{array}{ll} \forall X & S \| X \models \phi_1 \text{ and} \\ \forall X & S \| X \models \phi_2 \text{ and} \\ \ldots & \\ \forall X & S \| X \models \phi_n \end{array}$$

By partial model checking we obtain:

$$\begin{array}{ll} \forall X & X \models \phi'_1 \text{ and} \\ \forall X & X \models \phi'_2 \text{ and} \\ \ldots & \\ \forall X & X \models \phi'_n \end{array}$$

where for each $i$ from 1 to $n$, $\phi'_i = (\phi_i)_{//_S}$.

Let $Y_1, \ldots, Y_n$ be $n$ processes such that:

$$\begin{array}{ll} \forall X & Y_1 \triangleright_T X \models \phi'_1 \text{ and} \\ \forall X & Y_2 \triangleright_T X \models \phi'_2 \text{ and} \\ \ldots & \\ \forall X & Y_n \triangleright_T X \models \phi'_n \end{array}$$

It is possible to prove the following result.

**Lemma 4.3** *Let $\phi$ be a safety property, conjunction of $n$ safety properties,* i.e.*, $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$ where $\phi_1, \ldots \phi_n$ are safety properties. Let $Y_1, \ldots, Y_n$ be $n$ controller programs such that $\forall i$ such that $1 \leq i \leq n$ $Y_i \models \phi_i$. We have*

$$\forall X \quad Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X)))) \models \phi$$

This means that, once we have synthesized controller programs for enforcing several safety properties, we are able to enforce also the conjunction of them simply applying them successively. However, in this way, we apply the procedure for enforcing $n$ times. Instead we want apply it only one time to force the conjunction of formulas. For that reason we prove the following proposition.

**Proposition 4.12** *Let us consider the controller operator $\triangleright_T$. It is possible to find $Y_1, \ldots, Y_n$ controller programs such that. if $Y_1 \triangleright_T X \models \phi'_1, \ldots, Y_n \triangleright_T X \models \phi_n$ then $(Y_1 \triangleright_T \ldots \triangleright_T Y_n) \triangleright_T X \models \phi_1 \wedge \ldots \wedge \phi_n$.*

Hence, referring to the Formula (4.13), in order to find $Y$ we find $Y_1, \ldots, Y_n$ that enforce $\phi'_1, \ldots, \phi'_n$ respectively and we compose them as in Proposition 4.12. In this way we find $Y$ that force $\phi' = \phi'_1 \wedge \ldots \wedge \phi'_n$. According to Lemma 2.6 we have:

$$\forall X \quad Y \triangleright_T X \models \phi' \Leftrightarrow \forall X \quad S \| Y \triangleright_T X \models \phi$$

Hence we obtain a controller program $Y$ for $\phi$.

## 4.4 Synthesis of controller programs for parameterized systems

A parameterized system describes an infinite family of (typically finite-state) systems (see [18]). Instances of the family can be obtained by fixing parameters.

Let us consider a parameterized system $S = P_n$ defined by parallel composition of processes $P$, *e.g.*,

$$\underbrace{P \| P \| \ldots \| P}_{n}$$

The parameter $n$ represents the number of processes $P$ present in the system $S$.

**Example 4.1** *Consider a system with one consumer process $C$ and several producer processes $P$. Each process $P$ is defined $P \stackrel{def}{=} a.P$ where $a \in Act$, and the process $C$ is $\bar{a}.C$. Let us suppose that the system consists of $n$ producer and one consumer, then the entire system is $(P_n \| C) \backslash \{a\}$ and the processes communicate by synchronization on $\bar{a}$ and $a$ actions.*

Referring to the Formula (3.1) we may wish to have:

$$\forall n \quad \forall X \quad P_n \| X \models \phi \tag{4.14}$$

It is possible to note that in the previous equation there are two universal quantifications: The first one is on the number of instances of the process $P$, $n$, and the second one is on the possible behavior of the unknown agent.

In order to eliminate the universal quantification on the number of processes, firstly we define the concept of *invariant formula with respect to partial model checking for parallel operator* as follows.

**Definition 4.3** *A formula $\phi$ is said an* invariant with respect to partial model checking *for the system $P\|X$ if and only if $\phi \Leftrightarrow \phi_{//_P}$.*

It is possible to prove the following result.

**Proposition 4.13** *Given the system $P_k\|X$. If $\phi$ is an invariant formula for the system $P\|X$ then*

$$\forall X \quad (\forall n \quad P_n\|X \models \phi \quad \textit{iff} \quad X \models \phi)$$

In order to apply the theory developed in Section 3.2.1, we show a method to find the invariant formula. According to [18], let $\psi_i$ be defined as follows:

$$\psi_i = \begin{cases} \phi_1 & \text{if } i = 1 \\ \psi_{i-1} \wedge \phi_i & \text{if } i > 1 \end{cases}$$

where for each $i$, $\phi_i = \phi_{//_{P_i}}$.

By definition of $\psi_i$ and by Lemma 2.6, $\forall j$ such that $1 \leq j \leq i$ $(X \models \phi'_j) \Leftrightarrow X \models \psi_i$. Hence $X \models \psi_i$ means that $\forall j$ such that $1 \leq j \leq i$ $P_j\|X \models \phi$. We say that $\psi_i$ is said to be *contracting* if $\psi_i \Rightarrow \psi_{i-1}$. If $\forall i$ $\psi_i \Rightarrow \psi_{i-1}$ holds, we have a chain that is a said a *contracting sequence*. If it is possible to find the invariant formula $\psi_\omega$ for a chain of $\mu$-calculus formulas, that is also said *limit of the sequence*, then the following identity holds:

$$\forall X \quad (X \models \psi_\omega \Leftrightarrow \forall n \geq 1 \quad P_n\|X \models \phi) \tag{4.15}$$

Now we have a problem equivalent to the problem expressed in Formula 3.1. Hence, depending on which kind of property we are going to enforce, we consider one of the controller operators seen in Chapter 3 that forces each process to behave correctly. Then we apply the theory developed in previous section to synthesize a controller program for the chosen controller operator.

In some cases it could not be possible to find the limit of the chain. However there are some techniques that can be useful in order to find an approximation of this limit (see [18, 36]).

## 4.5 Synthesis of controller in a distributed system

In this section we show how it is possible to synthesize centralized and decentralized controller program for enforcing safety properties in a distributed system.

Referring to Section 3.2.4, in order to solve the problem expressed by the following formula:

$$\forall X_1, \ldots, X_n \quad C(X_1, \ldots, X_n) \models \phi \tag{4.16}$$

we use a controller context $\triangleright(Y, X)$. In Table 4.4 we recall the semantics definition of controller contexts we have defined in Section 3.2.4. We can follow a centralized approach or a decentralized one in order to enforce safety properties. In the following we show how synthesize controller programs for both these approaches.

$\triangleright_T$:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_T(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_T(Y',X')}$$

$\triangleright_S$:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_S(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_S(Y',X')} \qquad \frac{Y \xrightarrow{\overrightarrow{-a}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_S(Y,X) \xrightarrow{(\overrightarrow{-a},\overrightarrow{a})} \triangleright_S(Y',X')}$$

$\triangleright_I$:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_I(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_I(Y',X')} \qquad \frac{Y \not\xrightarrow{\overrightarrow{a}} Y' \quad Y \xrightarrow{\overrightarrow{+a.b}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_I(Y,X) \xrightarrow{(\overrightarrow{+a.b},\overrightarrow{a})} \triangleright_I(Y',X')}$$

$\triangleright_E$:

$$\frac{Y \xrightarrow{\overrightarrow{b}} Y' \quad X \xrightarrow{\overrightarrow{b}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{b},\overrightarrow{b})} \triangleright_E(Y',X')} \qquad \frac{Y \xrightarrow{\overrightarrow{-a}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{-a},\overrightarrow{a})} \triangleright_E(Y',X')}$$

$$\frac{Y \not\xrightarrow{\overrightarrow{a}} Y' \quad Y \xrightarrow{\overrightarrow{+a.b}} Y' \quad X \xrightarrow{\overrightarrow{a}} X'}{\triangleright_E(Y,X) \xrightarrow{(\overrightarrow{+a.b},\overrightarrow{a})} \triangleright_E(Y',X')}$$

Table 4.4: Semantics definition of controller contexts for enforcing safety properties.

**Centralized approach.** We synthesize a unique controller program $Y$ that enforces $\phi$ by monitoring the product of all component as a unique context. According to Section 3.2.4 we have to solve the following problem:

$$\exists Y \quad \forall X_1, \ldots, X_n \quad \triangleright (Y, X_1 \times \ldots \times X_n) \models \phi' \tag{4.17}$$

where $\phi' = \mathcal{W}(C, \phi)$. Now we want to synthesize $Y$.

Looking at the semantics definition of $\triangleright$, it is possible to prove that the following simulation relation holds:

$$\triangleright(Y, X) \prec Y$$

Let $\phi$ be in $Fr_\mu$ a safety properties. Since a proposition similar to the Proposition 4.1 holds also for contexts, in order to satisfy the Formula (3.19) it is sufficient to find a controller program $Y$ such that:

$$Y \models \phi'$$

Since $Y \in C_0^n$ it performs a vector of actions. Without lost of generality, we consider

that each transition is labeled by a vector as $\bar{a} = (a_1, \ldots, a_n)$ and we synthesize $Y$ by Theorem 2.2. So we are able to prove the following result.

**Theorem 4.5** *The problem described in Formula (4.17) is decidable.*

**Decentralized approach.** We synthesize several controller context $Y_i$ one for each unspecified components $X_i$ of the system. Hence we have to solve the following problem:

$$\exists Y_1, \ldots, Y_n \; \forall X_1, \ldots, X_n \; \triangleright (Y_1, X_1) \times \ldots \times \triangleright (Y_n, X_n) \models_{\times} \psi_1^k \times \ldots \times \psi_n^k \qquad (4.18)$$

In general we have to deal with a disjunction of product formulas and, in order to enforce it, we chose which product formula of the disjunction enforce. We can consider to have a saturated formula, we can solve the problem above by solving $n$ problems as the following form:

$$\exists Y_j \; \forall X_j \; \triangleright (Y_j, X_j) \models \psi_j^k$$

We are able to synthesize $n$ controller program able to enforce safety properties, *i.e.*, properties described as formulas of $Fr_\mu$, by Theorem 2.2. As a matter of fact we are in a case similar to the one described before: Instead of generating a controller program $Y \in C_0^n$ that control the product of $n$ components, we generate $n$ controller programs in $C_0^1$ such that, each of them, controls only one component $X_i$. So we are able to prove the following result.

**Theorem 4.6** *The problem described in Formula (4.18) is decidable.*

### 4.5.1 An example: *Chinese Wall* policy

In order to better explain the differences that exists between two different approaches, we show an example. Indeed we consider the *Chinese Wall* policy (see Section 2.2.2). Now we present how we enforce the *Chinese Wall* policy in a distributed system, a very common security property. The *Chinese Wall* policy says that we can choose to open or an element of the set $A$ or an element of a set $B$. If an element $x$ that belongs to the set $A$ is opened then we can open only element that does not belong to $B$ and viceversa, i.e. if we open an element $x$ of $B$ then we can open only elements that are not in $A$.We consider as distributed system $S = \|(X_1, X_2)$.

The Chinese Wall policy can be expressed by the formula $\phi = \phi_1 \vee \phi_2$ where

$$\begin{array}{lll} \phi_1 = \text{LET MAX} & W = & [open_A]W \wedge [open_B]\mathbf{F}\text{IN}W \\ \phi_2 = \text{LET MAX} & V = & [open_B]V \wedge [open_A]\mathbf{F}\text{IN}V \end{array}$$

Hence we calculate $\phi' = \mathcal{W}(\|, \phi) = \phi'_1 \vee \phi'_2$ as follows:

$\phi'_1 =$
$\text{LET MAX}W' = [(0, open_A)]W' \wedge [(open_A, 0)]W' \wedge [(0, open_B)]\mathbf{F} \wedge [(open_B, 0)]\mathbf{F})\text{IN}W'$
$\phi'_2 =$
$\text{LET MAX}V' = [(0, open_B)]V' \wedge [(open_B, 0)]V' \wedge [(0, open_A)]\mathbf{F} \wedge [(open_B, 0)]\mathbf{F}\text{IN}V'$

Now we synthesize a controller program for enforcing $\phi$ in both cases.

**Centralized control.** In this case we generate a controller program $Y \in C_0^2$ such that it is a model for the formula $\phi'$ and a controller program for $\triangleright(Y, X_1 \times X_2)$. We take

$$\begin{array}{lll} Y & = + & (Y_1, Y_2) \\ Y_1 & = + & ((0, open_A)^*, (open_A, 0)^*)^\dagger \\ Y_2 & = + & ((0, open_B), (open_B, 0)^*)^\dagger \end{array}$$

It is possible to note that such $Y$ at the beginning permits whatever possible behavior of unspecified components. Indeed both $X_1$ and $X_2$ are allowed to perform $open_A$ or $open_B$ action. However, after the first step, only one behavior is allowed. Let us consider, for instance, $X_1 = (open_A)^* \circ X_2$ and $X_2 = (open_B)^* \circ X_1$. In this case $Y_1$ becomes $((open_A, 0)^*)^\dagger$ and $Y_2 = ((0, open_B)^*)^\dagger$ since the other choice case never happens. Hence $Y = +(((open_A, 0)^*)^\dagger, ((0, open_B)^*)^\dagger)$. Let consider that the first step is performed by $X_2$ then we have the following derivation tree:

$$\frac{\triangleright \xrightarrow{(0, open_B)} \triangleright \quad \| \xrightarrow{open_B} \|}{\| \circ \triangleright \xrightarrow{open_B} \| \circ \triangleright}$$

Hence:

$$\|(\triangleright(Y, X_1 \times X_2)) \xrightarrow{open_B} \|(\triangleright(Y_2, X_1 \times X_1))$$

Looking at the transition rule of $\triangleright$ we can note that, at the beginning, both the possibility, executing the action $open_A$ as well as executing the action $open_B$, are allowed. Since the first step is performed by $X_2$, the action $open_B$ is done. Hence the controller program chooses the component $Y_2$ that allows only $open_B$ actions. However, after the transition, the target system can perform only action $open_A$ so the system halts.

**Decentralized control.** Here we want to describe how it is possible to apply the decentralized method for enforcing a policy.

It is easy to note that the formula $\phi'$ is not finite. Hence a finite decomposition for it may not exists. In this case it may be difficult to find a formula to enforce.

Moreover is that the formula $\phi'$ requires a strict interaction between processes. As a matter of fact, the behavior of the second context is conditioned by the behavior of the first one and viceversa. Indeed if the first context performs an action $open_A$ ($open_B$) the second cannot performs an action $open_B$ ($open_A$) and viceversa. For this reason it is not possible to find two independent controller operators that enforce the Chinese wall policy in a distributed way at run-time.

On the other hand, we could enforce the Chinese wall policy in a distributed way by establishing a priori that one of the two contexts must not perform any action. For instance we can consider the following two controller programs:

$$Y_1 = Nil \qquad \begin{array}{lll} Y_2 & = + & (Y_2', Y_2'') \\ Y_2' & = & (open_A^*)^\dagger \\ Y_2'' & = & (open_B^*)^\dagger \end{array}$$

In this way, the first unknown component cannot perform any action and the second has to respect the Chinese wall policy. For instance, let $X_1 = (open_A)^* \circ X_2$ and $X_2 = (open_B)^* \circ X_1$ the we have that $\triangleright(Nil, X_1)$ is equivalent to the context $Nil$ because does not perform any action. Hence the system is $\|(Nil, (\triangleright(Y_2, X_2)))$. Thus we have the following transition:

$$\|(Nil, (\triangleright(Y_2, X_2))) \xrightarrow[(0, open_B)]{(open_B)} \|(Nil, (\triangleright(Y_2'', X_1)))$$

since, according to the semantic definition of $\triangleright$, the transduction of $\triangleright(Y_2, X_2)$ is the following:

$$\triangleright(Y_2, X_2) \xrightarrow[(open_B, open_B)]{(open_B)} \triangleright(Y_2'', X_1)$$

The execution halts because the second contexts, by calling the first one, tries to performs an action $open_A$ that is forbidden.

### 4.5.2 Another example

Here we show another simple example in which also the distributed approach can be applied at run-time. In this way, we aim to underline which are the differences between the centralized and decentralized approach to make secure a distributed system.

Let $S$ be the system such that $S = \|(X_1, X_2)$ and let $\phi = [a][a]\mathbf{F}$.

First of all we apply the property transformer in order to find the weakest property that must be satisfied by the unspecified part of the system. Hence we calculate $\mathcal{W}(\|, \phi)$ and we obtain the following formula:

$$\phi' = [(0,a)]([(0,a)]\mathbf{F} \wedge [(a,0)]\mathbf{F}) \wedge [(a,0)]([(0,a)]\mathbf{F} \wedge [(a,0)]\mathbf{F})$$

Now we synthesize a controller program for enforcing $\phi$ in both cases.

**Centralized control.** In this case we generate a controller program $Y \in C_0^2$ such that it is a model for the formula $\phi'$ and a controller program for $\triangleright(Y, X_1 \times X_2)$. We take, for instance

$$Y = +((0,a)^* \circ Nil, (a,0)^* \circ Nil)$$

It is not difficult to note that $Y$ satisfies the formula $\phi'$.

In order to show how $\triangleright$ works we consider, for instance, two possible behaviors for the unspecified components and we show which are the tranductions of the system $\|(\triangleright(Y, X_1 \times X_2))$.

Let $X_1 = a^* \circ Nil$ and $X_2 = a^* \circ Nil$ be two possible behaviors of the unspecified components. Then, supposing that the second component performs the action $a$, we have:

$$\frac{\triangleright \xrightarrow[(0,a,0,a)]{(0,a)} \triangleright \quad \| \xrightarrow[(0,a)]{a} \|}{\| \circ \triangleright \xrightarrow{a} \| \circ \triangleright}$$

Hence:

$$\|(\triangleright(+((0,a)^* \circ Nil, (a,0)^* \circ Nil), X_1 \times X_2)) \xrightarrow{a} \|(\triangleright(Nil, X_1 \times Nil))$$

The execution halts when also $X_1$ wants to perform $a$, since it is not allowed because $\phi$ requires that two $a$ actions are not performed sequentially. It is not difficult to note that a similar scenario occurs if $X_1$ performs $a$. Moreover, there is also the possibility that both the components try to perform the $a$ action at the same time.

**Decentralized control.** In order to synthesize distributed controller programs we have to decompose the formula $\phi$ in such a way we can write it as a disjunctive product formula. This traduction is possible by exploiting the weak equivalences for decomposition properties recalled in Table 3.1. In particular we use the following $\mathbf{F} \Leftrightarrow \mathbf{T} \times \mathbf{F} \vee \mathbf{F} \times \mathbf{T}$ and $\mathbf{T} \Leftrightarrow \mathbf{T} \times \mathbf{T}$. After several steps, it is possible to write $\phi$ is the following weak equivalent way:

$$\begin{aligned}&[(0,a)]([(0,a)]\mathbf{F} \wedge [(a,0)]\mathbf{F}) \wedge [(a,0)]([(0,a)]\mathbf{F} \wedge [(a,0)]\mathbf{F})\\ &\Leftrightarrow ([a]\mathbf{F} \times [a]\mathbf{F}) \vee ([a]\mathbf{F} \times [a][a]\mathbf{F}) \vee ([a][a]\mathbf{F} \times [a]\mathbf{F})\end{aligned}$$

In this case, we want to synthesize two controller programs $Y_1$ and $Y_2$ s.t. $\triangleright(Y_1, X_1)$ and $\triangleright(Y_2, X_2)$. In the distributed case we do not enforce the global property but we choose which subformula of the disjunction we enforce. For instance we enforce $([a]\mathbf{F} \times [a][a]\mathbf{F})$. Hence we synthesize $Y_1$ s.t. it is model for $[a]\mathbf{F}$ and $Y_2$ s.t. it is model for $[a][a]$.

By exploiting a satisfiability properties we can consider $Y_1 = Nil$ and $Y_2 = a^* \circ Nil$. Let $X_1 = a^* \circ Nil$ and $X_2 = a^* \circ Nil$ and we obtain:

$$\|(\triangleright(Nil, a^* \circ Nil), \triangleright(a^* \circ Nil, a^* \circ Nil)) \xrightarrow{\overline{(a,a)}} \|(\triangleright(Nil, a^* \circ Nil), \triangleright(Nil, Nil))$$

Since the first component $\triangleright(Nil, a^* \circ Nil)$ is not allowed to perform any action and the second one ends, the whole system performs $a$ and halts.

By comparing these two approaches it is possible to note that in the first one, the controller program allowed more than one correct behavior, *i.e.*, either $X_1$ or $X_2$ could perform the first $a$ action. This is possible since it has a centralized view of what all the components do with respect to their allowed behavior. When one of the two executes the action $a$, the other should not do it after.

On the other hand, the decentralized controller programs are independent. For this reason we chose a priori which behavior we are going to enforce. We have chosen to allow the second component to do the $a$ action. In this way we do not consider all possible right behaviors of the components even if, at the end, also in this case, at the global level the system is secure.

## 4.6 Synthesis of Web services orchestrator

In this section we show how it is possible to apply our approach based on partial model checking, process algebra and satisfiability procedure for temporal logic, to deal with *Web*

*Services*.

In particular we put our attention on the problem of the synthesis of a Web Service *orchestrator* in a timed setting, *i.e.*, we want to find an orchestrator process that, given a network of services and a user's request, is able to manage the services in order to satisfy the request.

According to the Section 2.3.4, it is possible to model each service, described inBPEL, by a $timedCCS$ process.

We consider an orchestrator process as a monitor that coordinates and composes services in order to satisfy a user's request. We apply the same techniques used in [93, 98] and described in Section 4.1 and Section 4.2 to guarantee that a system is secure.

Let us consider to have a network of services made up of $n$ endpoints and that each of them provides a service.*i.e.*, we consider $P_1, \ldots, P_n$, $n$ finite-state processes that model the behavior of $n$ services of the considered network. Being $P_1, \ldots, P_n$ finite state processes, the satisfiability problem that we are going to solve is decidable.

Moreover we assume that sets of actions of processes $P_i$ are pairwise disjoint, *i.e.*, let $L_i$ and $L_j$ be the sets of actions respectively of $P_i$ and $P_j$ then $L_i \cap L_j = \emptyset$. This assumption guarantees that all possible synchronization between processes are established and coordinated by the orchestrator process.

Let $\phi$ be an equational $\mu$-calculus formula that expresses a possible request of an user. We want to find a process $\mathcal{O}$, that is the *orchestrator process*, that by managing $P_1, \ldots, P_n$ satisfies the request $\phi$.

In order to do this we define a process algebra operator, denoted by $\triangleright$, said *orchestrating operator* whose semantics definition is the following:

$$\frac{\mathcal{O} \xrightarrow{\tau} \mathcal{O}'}{\mathcal{O} \triangleright P \xrightarrow{\tau} \mathcal{O}' \triangleright P} \qquad \frac{\mathcal{P} \xrightarrow{\tau} \mathcal{P}'}{\mathcal{O} \triangleright P \xrightarrow{\tau} \mathcal{O} \triangleright P'} \qquad \frac{\mathcal{O} \xrightarrow{a} \mathcal{O}' \quad P \xrightarrow{a} P'}{\mathcal{O} \triangleright P \xrightarrow{a} \mathcal{O}' \triangleright P'}$$

$$\frac{\mathcal{O} \xrightarrow{tick} \mathcal{O}' \quad P \xrightarrow{tick} P'}{\mathcal{O} \triangleright P \xrightarrow{tick} \mathcal{O}' \triangleright P'}$$

By applying the last semantics rule we consider the elapsing of time. As a matter of fact, we consider the possibility that $P$ performs a $tick$ action. In this case the orchestrator process $\mathcal{O}$ permits it.

Hence we can formalize the composition problem as follows:

$$\exists \mathcal{O}\ \mathcal{O} \triangleright P \models \phi \tag{4.19}$$

where $P = P_1 \| \ldots \| P_n$.

We want to reduce the validity problem described in the Formula (4.19) to a satisfiability problem by exploiting the partial model checking function with respect to the orchestrating operator $\triangleright$. For this reason we define the partial evaluation function with respect to $\triangleright$ according to the operational semantics definition of the operator. Its definition is given in Table 4.5. The following proposition, similar to Lemma 2.6, holds.

$$
\begin{array}{lcl}
Z//_{\triangleright P} & = & Z_{\triangleright P} \\
(Z =_\sigma \phi D)//_{\triangleright P} & = & ((Z_{\triangleright P} =_\sigma \phi//_{\triangleright P})(D)//_{\triangleright P}) \\
[\alpha]\phi//_{\triangleright P} & = & \begin{cases} \bigwedge_{P \xrightarrow{\alpha} P'} [\alpha]\phi//_{\triangleright P'} & \text{if } P \xrightarrow{\alpha} P' \\ \mathbf{T} & \text{if } P \not\xrightarrow{\alpha} \end{cases} \quad \alpha \neq \tau \\
[\tau]\phi//_{\triangleright P} & = & [\tau](\phi//_{\triangleright P}) \wedge \bigwedge_{P \xrightarrow{\tau} P'} \phi//_{\triangleright P'} \\
\langle\alpha\rangle\phi//_{\triangleright P} & = & \begin{cases} \bigvee_{P \xrightarrow{\alpha} P'} \langle a\rangle\phi//_{\triangleright P'} & \text{if } P \xrightarrow{\alpha} P' \\ \mathbf{F} & \text{if } P \not\xrightarrow{\alpha} \end{cases} \quad \alpha \neq \tau \\
\langle\tau\rangle\phi//_{\triangleright P} & = & \langle\tau\rangle(\phi//_{\triangleright P'}) \vee \bigvee_{P \xrightarrow{\tau} P'} \phi//_{\triangleright P'} \\
\phi_1 \vee \phi_2//_{\triangleright P} & = & (\phi_1//_{\triangleright P}) \vee (\phi_2//_{\triangleright P}) \\
\phi_1 \wedge \phi_2//_{\triangleright P} & = & (\phi_1//_{\triangleright P}) \wedge (\phi_2//_{\triangleright P}) \\
\mathbf{T}//_{\triangleright P} & = & \mathbf{T} \\
\mathbf{F}//_{\triangleright P} & = & \mathbf{F}
\end{array}
$$

Table 4.5: Partial evaluation function for $\triangleright$ operator.

**Proposition 4.14** *Let $P$ and $Q$ be two finite state processes,*

$$Q \triangleright P \models \phi \textit{ iff } Q \models \phi//_{\triangleright P}$$

Hence, by using the partial evaluation function, we are able to evaluate the behavior of the composition of services directly into the request of the user. Moreover it permits to underline which is the behavior of the orchestrator in order to guarantee that the request is satisfied according to the semantics definition of the operator $\triangleright$.

In order to understand better how partial model checking with respect to $\triangleright$ operator works, we show a simple example.

**Example 4.2** *Let $\phi = [\alpha]\phi'$ be an user's request. We want to evaluate the formula $\phi$ with respect to the $\triangleright$ operator and a process $P$. According to the rule for the box modality in Table 4.5, the formula $\phi_{//\triangleright P}$ is satisfied by $\mathcal{O}$ if, whenever $P$ performs the action $\alpha$, also $\mathcal{O}$ performs the same action. This is taken into account by the first case of the formula,* i.e., $\bigwedge_{P \xrightarrow{\alpha} P'} [\alpha]\phi//_{\triangleright P'}$. *On the other hand, if $P$ does not performs $\alpha$ then the formula becomes always* $\mathbf{T}$.

Applying the partial model checking we are able to reduce the problem described in Formula 4.19 as follows:

$$\exists \mathcal{O} \quad \mathcal{O} \models \phi'$$

where $\phi' = \phi//_{\triangleright P}$.

According to Theorem 2.2, it is possible to find a model for a given equational $\mu$-calculus formula. Thus we synthesize an orchestrator process for a given request in a timed setting.

Since we are considering finite-state processes, the following result holds.

**Theorem 4.7** *The problem described in Formula (4.19) is decidable.*

### 4.6.1 An example

Let us suppose there is an user that want to organize a trip. Let us suppose that a such user makes the following request to a possible network of services:

*After booking an hotel I need to receive a confirmation before booking a flight.*

Let $Act_t = \{\texttt{b\_h}, \texttt{b\_f}, \texttt{b\_c}, \texttt{conf}, \tau, \ tick\}$ be the set of actions, where $\texttt{b\_h}$ permits to book an hotel, $\texttt{conf}$ is the confirmation from the hotel, $\texttt{b\_f}$ is the booking of the flight and $\texttt{b\_c}$ permits to reserve a car. By assuming that a possible confirmation can arrive immediately or after an amount of time, that we model by a $tick$ action, we model the user's request by an equational $\mu$-calculus formula $\phi$ as follows:

$$\phi = \langle \texttt{b\_h} \rangle (\langle \texttt{conf} \rangle \langle \texttt{b\_f} \rangle \mathbf{T} \vee \langle tick \rangle \langle \texttt{conf} \rangle \langle \texttt{b\_f} \rangle \mathbf{T})$$

This means that after booking the hotel the confirmation arrives immediately and the user books the flight or some time passes before the confirmation arrives and then the user books the flight.

Let us consider two processes $P_1$ and $P_2$ such that

$$\begin{aligned} P_1 &= \texttt{b\_h}.tick.\texttt{conf}.\mathbf{0} \\ P_2 &= \texttt{b\_f}.(\tau.\mathbf{0} + P_2') \\ P_2' &= \texttt{b\_c}.\mathbf{0} \end{aligned}$$

This means that $P_1$ allows to book an hotel and gives back the confirmation and $P_2$ allows to book a flight or to book a flight and reserve a car.

After the application of partial model checking we obtain $\phi //_{\triangleright P} = \phi'$ as follows:

$$\phi' = \langle \texttt{b\_h} \rangle \langle tick \rangle \langle \texttt{conf} \rangle \langle \texttt{b\_f} \rangle \mathbf{T}$$

Hence, as model for $\phi'$, we can consider the following process:

$$\mathcal{O} = \texttt{b\_h}.tick.\texttt{conf}.\texttt{b\_f}.\mathbf{0}$$

Looking at the semantics definition of $\triangleright$, the execution of $\mathcal{O} \triangleright P$ consists on the following sequence of transitions:

$$\begin{aligned} &\texttt{b\_h}.tick.\texttt{conf}.\texttt{b\_f}.\mathbf{0} \ \triangleright \ (\texttt{b\_h}.tick.\texttt{conf}\mathbf{0} \| \texttt{b\_f}.(\tau.\mathbf{0} + P_2')) \\ &\qquad\qquad \downarrow \texttt{b\_h} \\ &tick.\texttt{conf}.\texttt{b\_f}.\mathbf{0} \ \triangleright \ (tick.\texttt{conf}\mathbf{0} \| \texttt{b\_f}.(\tau.\mathbf{0} + P_2')) \\ &\qquad\qquad \downarrow tick \\ &\texttt{conf}.\texttt{b\_f}.\mathbf{0} \ \triangleright \ \texttt{conf}\mathbf{0} \| \texttt{b\_f}.(\tau.\mathbf{0} + P_2') \\ &\qquad\qquad \downarrow \texttt{conf} \\ &\texttt{b\_f}.\mathbf{0} \ \triangleright \ \mathbf{0} \| \texttt{b\_f}.(\tau.\mathbf{0} + P_2') \\ &\qquad\qquad \downarrow \texttt{b\_f} \\ &\mathbf{0} \ \triangleright \ \mathbf{0} \| (\tau.\mathbf{0} + P_2') \end{aligned}$$

At this time the process ends without booking the car. This is exactly what the user required. As a matter of fact the orchestrating operator does not consider additional services that are not required by the user, as, in this case, the reservation of a car.

## 4.7 Related work on synthesis

In [87] the authors provided a preliminary work in which there are presented several techniques for automatically synthesizing systems enjoying a very strong security property, *i.e.*, $SBSNNI$ (see [49]). This is also called $P\_BNDC$ in [54]. In both these works the authors did not deal with control theory.

In [124] the author has defined the notion of *enforcement mechanism* as a mechanism that work by monitoring a target system and terminating any execution that is about to violate the security policy being enforced. The policy that enforcement mechanisms can enforced are safety properties. Hence the author have defined *security automata* as finite state automata that are able to recognize safety properties. Successively, starting from the work of Schneider, Ligatti *et al.* in [19, 20] have analyzed the space of security policies that can be enforced by monitoring programs at runtime. Such programs are automata that examine the sequence of program actions and transform the sequence when it deviates from the specified policy. The simplest such automaton truncates the action sequence by terminating a program. Such automata are commonly known as security automata. In Chapter 3, we have shown how we are able to model the behavior of these automata by exploiting process algebras and logics. In this Chapter we have extended this result by showing how our approach permits us also to synthesize a controller program that effectively makes the system secure. The synthesis problem is not addressed nor in [124] neither in [19, 20].

The synthesis of controllers is a framework addressed also in other research areas (*e.g.*, see [7, 75, 122, 135]).

Our works on controller mechanisms starts from the necessity to make systems secure regardless the behavior of possible intruders, *i.e.*, we suppose that the system we consider works in parallel with a unknown components, that represent a possible malicious agent, and we have developed mechanisms to guarantee the system is secure whatever the behavior of the possible malicious agent is.

A lot of work has been done in order to study and analyze systems to guarantee the they satisfy certain security properties. In this chapter we have presented how the logical approach based on open system paradigm for the security analysis, in particular for the specification and verification (see [85]) can be extended also to synthesize mechanisms able to force a system to guarantee security properties.

In literature there are many works dealing with the sytnhesis problem. For instance, in [63, 119] the authors have showed mechanisms to synthesize safety properties. However they expressed safety properties by using linear time logic. They develop two method: *off-line* and *inline*. In particular they deal with *past time linear temporal logic*, denoted by $ptLTL$, in order to have a linear complexity. In fact every $ptLTL$ formula may be encoding by an $LTL$ one and the model checking for a $LTL$ formula has linear complexity. In particular the complexity of their algorithm is $\mathcal{O}(n \times m)$ where $n$ is the number of the states of the trace and $m$ is the size of the formula. In the *off-line* approach they create a monitor that runs in parallel with the executing program, receiving events from the running program, and checking on-the-fly that the formulas are satisfied. In this approach

the formulas to be checked are given in a separate specification. In the second *inline* approach, formulas are written as comments in the program test, and are then expanded into the code.

Many other approaches to the controller synthesize problem are based on game theory (see [5, 74, 82, 88]). As a matter of fact, different kinds of automata are used to model properties that must be enforced. Games are defined on the automata in order to find the structure able to satisfy the given properties. For instance in [5], the authors deal with the synthesis of controllers for discrete event systems by finding a winning strategies for a parity games. In this framework it is possible to extend the specifications of the supervised systems as well as the constraints on the controllers by expressing them in the modal $\mu$-calculus. In order to express un-observability constraints, they propose an extension of the modal mu-calculus in which one can specify whether an edge of a graph is a loop. This extended $\mu$-calculus still has the interesting properties of the classical one. The method proposed in this paper to solve a control problem consists in transforming this problem into a problem of satisfiability of a $\mu$-calculus formula so that the set of models of this formula is exactly the set of controllers that solve the problem. On the contrary, we synthesize controllers that work by monitoring only the possible un-trusted component of the system. Moreover they do not addressed any security analysis, *i.e.*, they synthesize controller for a given process that must be controlled. On the contrary we synthesize controllers that make the system secure for whatever behavior of unknown components. Our controller are synthesized without any information about the process they are going to control.

In [113, 117, 118] the authors developed a theory for the synthesis of the maximally permissive controller. The authors have proposed general synthesis procedure which always computes a maximal permissive controller when it exists. However they generate a maximal permissive controller given knowing the behavior of the process they are going to control. On the contrary, we do not nothing a priori on the possible behavior of a possible malicious agent whose behavior we want to control.

**Related work on synthesis of controller for distributed systems**

In Section 4.5, we have proposed a framework based on contexts and logic to deal with the problem of the synthesis of controllers for distributed systems. As a matter of fact, we have proposed two different ways to enforce security properties also when we are treating systems in which more then one components is unspecified.

In [10] the problem of the synthesis of controllers is studied. They start from their previous work, [11], in which they deal with the decentralized control problem of several communicating supervisory controllers, each with different information, that work in concert to exactly achieve a given legal sub-language of the uncontrolled system's language model. In [10] the author presents a procedure for finding an optimal communication policy, if one exists, for the a class of finite controllers. In our work, in addition to explicit un-trusted components, we deal with the problem of the distributed control but we consider independent controllers, *i.e.*, we present a method to synthesize decentralized

controllers that work independently one each other on different unspecified components of the system in such a way the whole system is secure.

It is worth also to mention approaches that directly try to build correct systems (rather than controlling potentially incorrect ones as we do). For instance in [37] the authors present an automatic synthesis procedure for distributed system having a flexible specification language and a reasonable computational complexity. They use *asynchronous automata*.

**Related work on synthesis of Web Services orchestration**

In the literature a lot of works deal with web services composition through formal methods. For instance several works deal with a possible modeling of by process algebras (see [8, 9, 31, 47, 123]) or by automata (see [115]).

In [14, 16] the authors have developed a static approach to deal with the composition of web services problem by the usage of *plans*. In particular they use a distributed, enriched $\lambda$-calculus for describing networks of services. Both, services and their clients, can protect themselves, by imposing security constraints on each other's behavior. Then, service interaction results in a *call-by-property* mechanism (see [15]), that matches the client requests with services.

The planning approach is followed also by Pistore *et al.* (see *e.g.*, [109, 110]) in order to generate an orchestrator. As a matter of fact, the authors have proposed a novel planning framework for the automated composition of Web Services in which, given a set of BPEL abstract specifications of published Web Services, and given a composition requirement, they generate automatically a BPEL concrete process that interacts asynchronously with the published services. Basically they compose all services and then, after building all possible plans, they extract the plan that satisfies the user's request.

In Section 4.6, we treated the problem of the automatic composition of services by exploiting a different approach with respect to the previous cited works. Our approach permits us to treat the problem also in a timed setting, topic that is not addressed in [109, 110]. Our approach is general because we can define other process algebra orchestrating operator and, by defining a partial model checking function according to the operational semantics definition of such operators, we can combine services in different ways.

Also Zavattaro *et. al* deals with the problem of composition on services. They have studied choreography more than orchestration. They have introduced a formal model for representing choreography. Their model is based on a *declarative part* and on a *conversational* one. The declarative part of their choreography formal model is based on the concept of *role* that represents the behavior that a participant has to exhibit in order to fulfill the activity defined by the choreography. Each role can store variables and exhibit operations.

In [30] the authors have formalized the concept of orchestrator as a process, associated to an identifier, that is able to exchange information, represented by variables, with other processes. This model takes inspiration form the abstract non-executable fragment of BPEL and abstracts away from variables values focussing on data-flow. Orchestrators

are executed on different locations, thus they can be composed by using only the parallel operator ($\|$). Processes can be composed in parallel, sequence and alternative composition. Communication mechanisms model Web Services *One-Way* and *Request-Response* operations. In our approach the communication between services is managed by the orchestrator process that permits interactions between several services. By partial model checking we evaluate the behavior of the composition in the request formula and the orchestrator process is a monitor that guarantees the composition behaves according the request. In this way we can synthesize the orchestrator process as a model of the resulting formula.

No one of these papers treat the synthesis of orchestrator problem in a timed setting. In the literature there are some works on modeling a timed BPEL with formal method. For instance, in [71] the authors propose the *Web Service Timed Transition System* model, which adopts the formalism of timed automata for capturing the specific aspects of Web Service domain. In this formalism, the fact that the operation takes certain amount of time is represented by time increment in the state, followed by the immediate execution of the operation. Intuitively, WSTTS is a finite-state machine equipped with set of clock variables. The values of these clock variables increase with the elapsing of time. Thus a Web Service composition is represented as a network of several such automata, where all clocks progress synchronously. The semantic of WSTTS is defined as a labeled transition system, where either the time passes or a transition from one state to another immediately takes place. In [21, 22] the authors have discussed the augmentation of business protocols with specifications of temporal abstractions, focusing in particular on problems related to compatibility and replaceability analysis. In [43] the authors, firstly, have defined a timed automata semantics for the *Orc* language, introduced in order to support a structured way of orchestrating distributed web services. *Orc* is intuitive because it offers concise constructors to manage concurrent communication, time-outs, priorities, failure of sites or communication and so forth. The semantics of *Orc* is also precisely defined. Timed automata semantics is semantically equivalent to the original operational semantics of *Orc*. In [76] the authors introduce COWS, *calculus for orchestration of web services*, as a new foundational languages for service oriented computing, whose design has been influenced by BPEL. It combines in an original way a number of ingredients borrowed from process calculi.

However all these papers deal with the modeling of web services in a timed setting but they do not treat the problem of the synthesis of an orchestrator process.

# Chapter 5

# A Tool for the Synthesis

In this chapter we describe the tool that we have implemented in order to automatically generate controller programs for enforcing safety properties. As a matter of fact we have developed a tool for generating controller program for a chosen controller operator for safety properties, *i.e.*, one among the controller operators $\triangleright_T$, $\triangleright_S$, $\triangleright_I$ and $\triangleright_E$ defined in Section 3.2.1.

The tool consists of two main modules, going after the theoretical approach described in the previous two chapters. The first one is the *MuDiv* tool developed by Nielsen and Andersen, that implements the partial model checking function for process algebra operators (see [2, 3]) and a second one is the *Synthesis* module implemented in O'Caml 3.09 (see [78]).

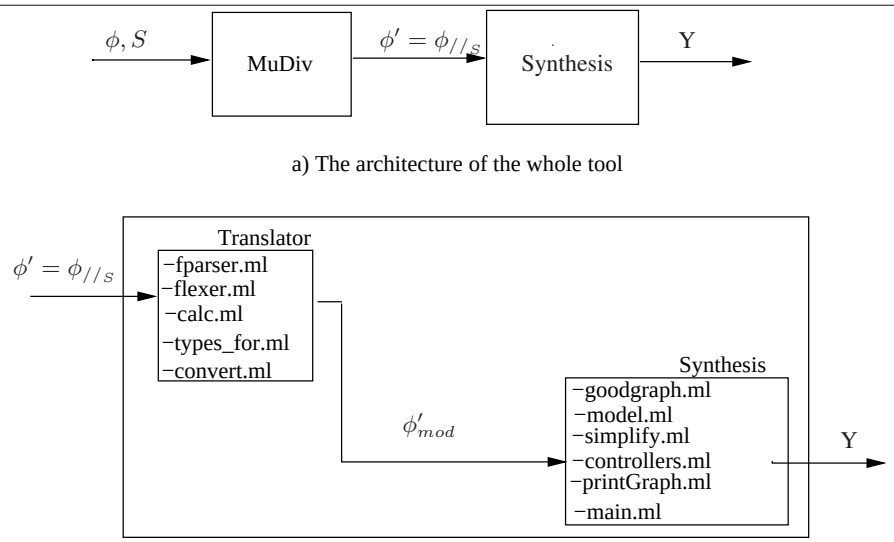
The *Mudiv* tool takes in input a system $S$ and a formula of equational $\mu$-calculus, $\phi$, and calculate $\phi' = \phi_{//_S}$ that is the partial evaluation of $\phi$ with respect to the system $S$.

The second part implements a satisfiability procedure for formulas in that described safety porpoerties (see Section 4.1) referring to the satisfiability procedure developed by Walukiewicz in [131] for a modal $\mu$-calculus formula. In particular it generates a model for $\phi'$, that is the output of the *MuDiv* tool. This model is also a controller program for a one of the four controller operators $\triangleright_T$, $\triangleright_S$, $\triangleright_I$ and $\triangleright_E$. It is possible to note that if $\phi$ is in safety properties, *i.e.*, a formula without $\mu$ fixpoint, also $\phi'$ is a formula of the same kind (see Table 2.14).

## 5.1 Synthesis tool

Now we describe more in detail the architecture of our implementation. As we have already said, it takes in input a system $S$ and a formula $\phi$ and gives in output a process $Y$, described as a labeled graph, that is a model for $\phi'$, the formula obtained by the partial evaluation of $\phi$ by $S$. According to the theory developed in previous chapter a such $Y$ guarantees $S \| (Y \triangleright X)$ satisfies $\phi$ whatever $X$ is.

The tool is made up of two main parts (see Figure 5.1.a)): The first part implements the partial model checking function; the second one, by referring the satisfiability procedure,




a) The architecture of the whole tool

b) A zoom of the Synthesis module

Figure 5.1: Architecture of the tool.

generates a process $Y$. In particular, it permits to obtain a controller program $Y$ for each controller operators $\triangleright_{\mathbf{K}}$.

In Figure 5.1 there is a graphical representation of the architecture of the whole tool that we explain in more detail in the following section.

### 5.1.1 Architecture of the tool

As we have already said, the tool is made up of two main parts: The *MuDiv* module and the *Synthesis* module.

#### *MuDiv* tool

The first module of our tool consists in the *MuDiv* module. It is a tool for verifying concurrent systems. It is based on the technique of partial model checking described in [3]. The technique uses the equational $\mu$-calculus to express the modal requirements and parallel composition of finite labeled transition systems to construct the model.

It has been developed in C++ by J.B. Nielsen and H.R. Andersen. The result is a non interactive batch program, where the input is provided as one or more input files, describing the model and the requirements. The output is the result of the model check and it is presented on the standard output or written to a file.

### *Synthesis* module

The second module of our tool is the *Synthesis* one. It is able to build a model for a given modal $\mu$-calculus formula by exploiting the satisfiability procedure. It is developed in O'caml 3.09 (see [78]) and it is described better in Figure 5.1.b) in which we can see that it consists of two submodules: the *Translator* and the *Synthesis*.

**The Translator module.** It manages the formula $\phi'$, output of the *MuDiv* module in order to obtain a formula that can be read from the Synthesis module. It "translates" $\phi'$ from an equational to a modal $\mu$-calculus formula. This translation is necessary because the Walukiewicz's satisfiability procedure was developed for modal $\mu$-calculus formulas instead the partial model checking was developed for equational $\mu$-calculus ones. It is important to underline that we do not implement the satisfiability procedure described by Walukievicz for all the $\mu$-calculus formulas. Anyway our implementation is referred to it.

The Translator module consists in several functions:

- `fparser.ml` and `flexer.ml` that permit to read the *MuDiv* output file and analyze it as input sequence in order to determine its grammatical structure with respect to our grammar.
- The function `calc.ml` calls `flexer.ml` and `fparser.ml` on a specified file. In this way we obtain an equational $\mu$-calculus formula $\phi'$ according to the type that we have defined in
- `type_for.ml`. The last function, `convert.ml`, translates the equational $\mu$-calculus formula $\phi'$ in the modal one $\phi'_{mod}$.

**The Synthesis module.** It implements a satisfiability procedure for safety properties, referred to satisfiability procedure described by Walukievicz in [131].

Given a modal $\mu$-calculus formula, $\phi'_{mod}$, we build a graph by following the set of axioms of the satisfiability procedure. For that reason we define the `type graph` as a list of triple $(n, a, n) \in GNode \times Act \times GNode$ where $GNode$ is the set of graph nodes. Each node of the graph represents a state $L(n)$ of the graph. Each node is characterized by the set of formulas that it satisfies.

In `model.ml` we build the entire graph for the given formula $\phi'_{mod}$. It takes as input a pair in $GNode \times Graph$ and, in a recursive way, builds the graph. Referring to [131]. we check if the graph that we have generated is effectively a model or a refutation for $\phi'_{mod}$ by using the function `goodgraph.ml`. This function takes in input a graph and gives back the boolean value TRUE if the graph is a model, FALSE otherwise and it halts. These two functions, `model.ml` and `goodgraph.ml`, work in pair in order to find a graph in which $\phi'_{mod}$ is satisfied. At the beginning we give in input a node labeled by $\phi$ and `Empty_Graph`, that represents the empty graph. Then, in a recursive way, we build the graph by checking it at each step by applying the function `goodgraph. ml`. It is important to note that the graph that we generate has some transitions that are labeled by

an action and some transition that come from the semantics of logical operations. If we are able to build the entire graph we use the function `simplify.ml` to extract exactly the process that is a model for $\phi'_{mod}$. Such process consists in the graph in which all nodes that are linked by logical operation are considered as a single node. In this way at the end we obtain a labeled transition system that represents a process. Such process is a model for $\phi'_{mod}$.

In order to synthesize a process $Y$ that is a model of $\phi'_{mod}$ as well as a controller program for a chosen controller operator, we have implemented the function `controllers.ml`. By using this function we relabel $Y$ according with the controller operator we want to use as it is prescribed by Proposition 4.2. In this way we obtain four different processes $Y = Y[f_T]$, because $f_T$ is the identity function on $Act$, $Y[f_S]$, $Y[f_I]$ and $Y[f_E]$.

In this submodule there are other functions:

- `printGraph.ml` that permits to print the graph as a sequence of nodes, each of them labeled by a list of formulae, connected one each other by arrows labeled by an action.

- The function `main.ml` that calls all the other functions and permits to create the executable file (`.exe`).

### Performance

For our experiments we have used a pc with a CPU Intel core duo T2600 2.16GHz, 1GB RAM and an operative system linux Fedora Core 6 kernel 2.6.19.1.

We have observed the behavior of the Synthesis module, *i.e.*, we have analyzed the performance only of this module because it is the part of the tool that effectively generates the controller program.

We have tested several formulas with different dimension and we have noticed that the amount of time spent from the machine to generate the monitors is directly proportional to the dimension of the formulas we have to enforce, *i.e.*, the amount of spent time increases with the size of the formula. The results we have observed are summarize in Table 5.1.

| Policy | Size | User time | System time |
|---|---|---|---|
| *Only action $a$ are allowed* | 7 | 0m0.005s | 0m0.001s |
| *It isn't allowed open a new file while another file is open* | 16 | 0m0.007s | 0m0.004s |
| *Chinese Wall* | 16 | 0m0.006s | 0m0.000s |
| *It isn't allowed performing three $open$ action sequentially* | 25 | 0m0.021s | 0m0.008s |

Table 5.1: Synthesis module experiments results.

### 5.1.2 A case study

In order to explain better how our tool works we present an example in which a system must satisfy a safety property. We generate a controller program for each of the four controllers defined in Section 3.2.1.

Let $S$ be a system. We suppose that all users that work on $S$ have to satisfy the following rule:

*You cannot open a new file while another file is open.*

It can be formalized by an equation system $D$ as follows:

$$Z_1 =_\nu [\tau]Z_1 \wedge [\mathtt{open}]Z_2$$
$$Z_2 =_\nu [\tau]Z_2 \wedge [\mathtt{close}]Z_1 \wedge [\mathtt{open}]F$$

#### Truncation

We halt the system if an user try to open a file while another is already open. In this case we generate a controller program $Y$ for $Y \triangleright_T X$ and we obtain:

$$Y = \mathtt{open.close}.Y$$

$Y$ is a model for $D$.

In order to show how it works as controller program for $Y \triangleright_T X$ we suppose to have a possible user $X$ that tries to open two different files. Hence $X = \mathtt{open.open.}\mathbf{0}$. Applying $Y \triangleright_T X$ we obtain:

$$Y \triangleright_T X =$$
$$\mathtt{open.close}.Y \triangleright_T \mathtt{open.open.}\mathbf{0} \xrightarrow{\mathtt{open}} \mathtt{close}.Y \triangleright_T \mathtt{open.}\mathbf{0}$$

Since $Y$ and $X$ are going to perform a different action, i.e. $Y$ is going to perform `close` while $X$ is going to perform `open`, the whole system halts.

#### Suppression

We suppose to decide to suppress any possible `open` action that can violate the property $D$. In this case we generate a controller program $Y$ for the controller $Y \triangleright_S X$. We obtain:

$$Y = {}^{-}\mathtt{open}.Y + \mathtt{open}.Y'$$
$$Y' = {}^{-}\mathtt{open}.Y' + \mathtt{close}.Y$$

Let we suppose to be in the same scenario described for the previous operator. Let $X$ be a user that tries to open two different files. Hence $X = \mathtt{open.open.}\mathbf{0}$. Applying $Y \triangleright_S X$ we obtain:

$$Y \triangleright_S X = {}^{-}\mathtt{open}.Y + \mathtt{open}.Y' \triangleright_S \mathtt{open.open.}\mathbf{0}$$
$$\xrightarrow{\mathtt{open}} {}^{-}\mathtt{open}.Y' + \mathtt{close}.Y \triangleright_S \mathtt{open.}\mathbf{0} \xrightarrow{\tau} Y' \triangleright_S \mathbf{0}$$

The whole system halts again because, even if a wrong action is suppressed, this controllers cannot introduce right actions.

**Insertion**

Let $Y$ be a controller program for the controller $Y \triangleright_I X$. We obtain:

$$Y = {}^{+}\texttt{open.close.open}.Y + \texttt{open}.Y'$$
$$Y' = {}^{+}\texttt{open.close.open}.Y' + \texttt{close}.Y$$

We consider $X$ that tries to open two different files. Hence $X = \texttt{open.open}.\mathbf{0}$. We obtain:

$$\begin{array}{l} Y \triangleright_I X = \\ {}^{+}\texttt{open.close.open}.Y + \texttt{open}.Y' \triangleright_I \texttt{open.open}.\mathbf{0} \\ \xrightarrow{\texttt{open}} {}^{+}\texttt{open.close.open}.Y' + \texttt{close}.Y \triangleright_I \texttt{open}.\mathbf{0} \\ \xrightarrow{\texttt{close}} \texttt{open}.Y' \triangleright_I \texttt{open.}\mathbf{0} \xrightarrow{\texttt{open}} Y' \triangleright_I \mathbf{0} \end{array}$$

We can note the $Y$ permits $X$ to perform the first action open. Then it checks that $X$ is going to perform another open by the action ${}^{+}\texttt{open}$. Hence $Y$ inserts an action close. After this action it permits $X$ to perform the action open. Since $X$ does not perform any another actions the whole system halts.

**Edit**

We consider to apply the controller operator $Y \triangleright_E X$. The controller program that we generate is the following:

$$Y = {}^{-}\texttt{open}.Y + {}^{+}\texttt{open.close.open}.Y + \ \texttt{open}.Y'$$
$$Y' = {}^{-}\texttt{open}.Y' + {}^{+}\texttt{open.close.open}.Y' + \texttt{close}.Y$$

We suppose again that $X = \texttt{open.open}.\mathbf{0}$. We have:

$$\begin{array}{l} Y \triangleright_E X = \\ {}^{-}\texttt{open}.Y + {}^{+}\texttt{open.close.open}.Y + \ \texttt{open}.Y' \triangleright_E \\ \triangleright_E \texttt{open.open}.\mathbf{0} \xrightarrow{\texttt{open}} \\ {}^{-}\texttt{open}.Y' + {}^{+}\texttt{open.close.open}.Y' + \texttt{close}.Y \triangleright_E \\ \triangleright_E \texttt{open}.\mathbf{0} \xrightarrow{\texttt{close}} \texttt{open}.Y' \triangleright_E \triangleright_E \texttt{open.}\mathbf{0} \xrightarrow{\texttt{open}} Y' \triangleright_E \mathbf{0} \end{array}$$

Also in this case, after the first action open, $Y$ checks if $X$ is going to perform another open by the action ${}^{+}\texttt{open}$ and then it inserts the action close in order to satisfy the property $D$. Then it permits to perform another open action.

# Chapter 6

# Conclusions and future work

In previous chapters we have illustrated some results towards a uniform theory for enforcing security properties. In particular, we have extended a framework based on process calculi and logical techniques, that have been shown to be suitable to model and verify several security properties, to tackle also synthesis problems of secure systems.

Using a framework for the specification and verification of security properties, we deal with the *synthesis* of secure systems, by showing:

1. several semantics definitions of process algebra controller operators;
2. how it is possible to synthesize controller programs for such controller operators.

Here, we summarize the results of this thesis.

**Definition of enforcing mechanism.** The first thread regards the run-time enforcement of security properties. Our approach is based on the open system paradigm. A possible intruder, or a malicious agent, is modeled as an unknown component of the system. Having to analyze a partially specified system, we wonder if there exists a method to guarantee that such an open system behaves correctly, *i.e.*, according to a given security property, whatever the behavior of the possible malicious component is.

To fulfill run-time enforcement, we propose two different mechanisms:

**Process algebra controller operators.** We define distinguished process algebra controller operators $\triangleright$ for the enforcement of safety properties and information flow properties.

These controller operators work by only monitoring the possible un-secure components, because the known part of the system is evaluated into the formula by using the partial model checking technique. This is an advantage of our approach because it could not be always possible to monitor the whole distributed architecture, while it could be possible for some of its components. In particular, we would like to have a method that only constraints un-trusted

components, *e.g.*, downloaded applets. Indeed, often not all the system needs to be checked, or it is simply not convenient to check it as a whole.

Moreover, we define controller contexts to deal with distributed systems with more than one unknown component. We develop two mechanisms of enforcement: a centralized mechanism and a decentralized one.

**On-line partial model checking.** This method uses the idea that at run-time we can consider only the execution trace of the unknown component. For that reason, at each step of the computation, we can evaluate, by partial model checking with respect to the prefix operator, if the action that the target is going to perform violates the security of the system or not.

**Synthesis of controller program for the given controller operators.** The second thread regards the possibility to generate a process guaranteeing that our system is secure whatever the behavior of its unknown component is. Our approach permits to treat several problems by using an unique framework. As a matter of fact, we synthesize controller programs for enforcing safety properties and information flow properties. Also, we are able to deal with systems in a timed setting and with parameterized systems. Finally, we treat the problem of properties composition.

We show that it is possible to use a uniform way, based on few concepts of concurrency and temporal logic theory for specifying, verifying and synthesizing secure systems.

By using the same approach based on open systems, we solve the problem of finding a possible implementation of a controller program, that, by monitoring the target, replaces the unknown component, in such a way that the whole system is secure.

For each controller operator $\triangleright$ we study the behavioral equivalence that holds between the process $Y \triangleright X$ and the controller program $Y$. According to a given operator $\triangleright$, we are able to enforce safety properties or information flow properties. It is worth noticing that controller operators for safety properties ($\triangleright_{\mathbf{K}}$) are less strict than controller operators for information flow ($\triangleright^*$). As a matter of fact, $Y \triangleright_{\mathbf{K}} X \preceq Y$, on the contrary $Y \triangleright^* X \approx Y$. This means that, also according to the semantics definition of these operators, whenever we are going to enforce an information flow property, the behavior of the target is almost completely covered by the behavior of the controller program.

In both cases it is possible to replace $Y \triangleright X$ with $Y$. Thus, according to the security property we are considering, we reduce our original problem to a validity problem. In this way, by applying a satisfiability procedure, we obtain a controller program $Y$ that is a model for the formula we want to enforce and a controller program for the controller operator we have chosen. However, the satisfiability problem for $\mu$-calculus formulas is decidable in exponential time in the dimension of the formula.

Moreover, we solve the synthesis problem in several scenarios.

- We synthesize controller contexts to deal with distributed systems with more than one unknown component, for both centralized and decentralized mechanisms.

  In the first case, we generate a unique controller program that monitors all the unspecified components. In the second case, we synthesize a controller program for each unspecified components of the system in such a way that, by composing the properties enforced by each controller program, we obtain the security property that the global system must satisfy. Here, we define controller contexts for enforcing safety properties, as future work we are working to extend this results also to information flow properties.

- By using our approach, we are able to synthesize also orchestrator processes for composing web services. We define a process algebra operator for orchestrating processes and we define the partial model checking function with respect to the semantics definition of this operator. As future work, we aim to apply our approach to allow a secure composition of web services.

**The implementation phase.** We present a tool for the synthesis of a controller program. The tool merges our implementation of a satisfiability procedure based on the Walukiewicz's algorithm and the partial model checking technique. In particular, starting from a system and a formula describing a security property, the tool generates a process that, by monitoring a possible un-trusted component, guarantees that the system with an unspecified component satisfies the required formula whatever the target is.

# Bibliography


[1] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, 1994.

[2] H. R. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, 1993.

[3] H. R. Andersen. Partial model checking (extended abstract). In *Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, K. L. F. Leymann, D. Roller, D. Smith, S. Thatte, I. Trickovic, , and S. Weerawarana. Specification: Business process execution language for web services version 1.1. 2003.

[5] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1):7–34, 2003.

[6] E. Asarin and C. Dima. Balanced timed regular expressions. *Electr. Notes Theor. Comput. Sci.*, 68(5), 2002.

[7] E. Badouel, B. Caillaud, and P. Darondeau. Distributing finite automata through petri net synthesis. *Journal on Formal Aspects of Computing*, 13:447–470, 2002.

[8] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. *Electr. Notes Theor. Comput. Sci.*, 105:21–36, 2004.

[9] L. Bao, W. Zhang, and X. Zhang. Describing and verifying web service using ccs. *pdcat*, 0:421–426, 2006.

[10] G. Barrett and S. Lafortune. On the synthesis of communicating controllers with decentralized information structures for discrete-event systems. In *Proceedings of the 37th IEEE international conference on Decision and Control*, volume 3, pages 3281–3286, December 1998.

[11] G. Barrett and S. Lafortune. Decentralized supervisory control with communicating controllers. *IEEE Trans. Automatic Control*, 45(9):1620–1638, 2000.

[12] M. Bartoletti, P. Degano, and G. L. Ferrari. Checking risky events is enough for local policies. In M. Coppo, E. Lodi, and G. M. Pinna, editors, *ICTCS*, volume 3701 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2005.

[13] M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *CSFW*, pages 211–223. IEEE Computer Society, 2005.

[14] M. Bartoletti, P. Degano, and G. L. Ferrari. Plans for service composition. In *Workshop on Issues in the Theory of Security (WITS)*, 2006.

[15] M. Bartoletti, P. Degano, and G. L. Ferrari. Security issues in service composition. In *Invited talk at (FMOODS)*, 2006.

[16] M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. In *Proc. 19th Computer Security Foundations Workshop (CSFW)*, 2006.

[17] D. Basin, E.-R. Olderog, and P. E. Sevinç. Specifying and analyzing security automata using csp-oz. In *AsiaCCS 2007*. ACM, ACM, March 2007.

[18] S. Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. In *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 315–330, Warsaw, Poland, April 2003. Springer.

[19] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In I. Cervesato, editor, *Foundations of Computer Security: proceedings of the FLoC'02 workshop on Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002. DIKU Technical Report.

[20] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[21] B. Benatallah, F. Casati, J. Ponge, and F. Toumani. Compatibility and replaceability analysis for timed web service protocols. In *BDA*, 2005.

[22] B. Benatallah, F. Casati, J. Ponge, and F. Toumani. On temporal abstractions of web service protocols. In O. Belo, J. Eder, J. F. e Cunha, and O. Pastor, editors, *CAiSE Short Paper Proceedings*, volume 161 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[23] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[24] G. Bhat and R. Cleaveland. Efficient model checking via the equational $\mu$-calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 1996. IEEE Computer Society Press.

[25] B. Bloom. Structured operational semantics as a specification language. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 107–117, San Francisco, California, 1995. ACM Press.

[26] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.

[27] J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction*. Handbook of Process Algebra. Elsevier, 2001.

[28] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[29] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 45–59, London, UK, 1997. Springer-Verlag.

[30] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2005.

[31] J. Cámara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing wsbpel business processes using process algebra. *Electr. Notes Theor. Comput. Sci.*, 154(1):159–173, 2006.

[32] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete event processes with partial observations. *IEEE, Trans. Automat. Contr.*, 33:249–260, 1988.

[33] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 410–422, London, UK, 1993. Springer-Verlag.

[34] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In K. G. Larsen and A. Skou, editors, *CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 1991.

[35] F. Corradini, D. D'Ortenzio, and P. Inverardi. On the relationships among four timed process algebras. *Fundam. Inform.*, 38(4):377–395, 1999.

[36] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static-analysis of programs by construction or approximation offixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[37] A. Ştefănescu. Automatic synthesis of distributed systems. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 315, Washington, DC, USA, 2002. IEEE Computer Society.

[38] M. Dam. CTL$^*$ and ECTL$^*$ as fragments of modal $\mu-$calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[39] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[40] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes (concurrent programming). *Theoretical Computer Science*, 34(1-2):83–133, 1984.

[41] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[42] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[43] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of computation orchestration via timed automata. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2006.

[44] J. Elmqvist, S. Nadjm-Tehrani, and M. Minea. Safety interfaces for component-based systems. In R. Winther, B. A. Gran, and G. Dahll, editors, *SAFECOMP*, volume 3688 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2005.

[45] J. E.M. Clarke, O. Gumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[46] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the $\mu$-calculus and its fragments. *Theor. Comput. Sci.*, 258(1-2):491–522, 2001.

[47] A. Ferrara. Web services: a process algebra approach. In M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors, *ICSOC*, pages 242–251. ACM, 2004.

[48] R. Focardi. *Analysis and Automatic Detection of Information Flows in Systems and Networks*. PhD thesis, Department of Computer Science, University of Bologna, 1998.

[49] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[50] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 27:550–571, 1997.

[51] R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). 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 331–396, London, UK, 2001. Springer-Verlag.

[52] R. Focardi, R. Gorrieri, and F. Martinelli. Classification of security properties - part ii: Network security. In *FOSAD*, volume 2946 of *Lecture Notes in Computer Science*, pages 139–185, 2004.

[53] R. Focardi and R.Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1):5–33, 1997.

[54] R. Focardi and S. Rossi. Information flow security in dynamic contexts, 2002.

[55] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM Press.

[56] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, London, UK, 1987. Springer-Verlag.

[57] J. H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.

[58] S. Gnesi, G. Lenzini, and F. Martinelli. Applying generalized non deducibility on compositions (GNDC) approach in dependability. *Electr. Notes Theor. Comput. Sci.*, 99:111–126, 2004.

[59] S. Gnesi, G. Lenzini, and F. Martinelli. Logical specification and analysis of fault tolerant systems through partial model checking. *International Workshop on Software Verification and Validation (SVV), ENTCS.*, 2004.

[60] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proc. of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1982.

[61] R. Gorrieri, R. Lanotte, A. Maggiolo-Schettini, F. Martinelli, S. Tini, and E. Tronci. Automated analysis of timed security: a case study on web privacy. *Int. J. Inf. Sec.*, 2(3-4):168–186, 2004.

[62] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[63] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356, London, UK, 2002. Springer-Verlag.

[64] L. Heerink and E. Brinksma. Validation in context. In *PSTV*, pages 221–236, 1995.

[65] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[66] M. Hennessy and T. Regan. A temporal process algebra. In *FORTE '90: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 33–48. North-Holland, 1991.

[67] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[68] C. Jard and T. Jéron. On-line model checking for finite linear temporal logic specifications. In *Automatic Verification Methods for Finite State Systems*, pages 189–196, 1989.

[69] C. Jard, T. Jeron, J. C. Fernandez, and L. Mounier. On-the-fly verification of finite transition systems. Technical Report RR-1861, IRISA, 1996.

[70] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[71] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in-web service compositions. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, pages 840–846, Washington, DC, USA, 2006. IEEE Computer Society.

[72] D. Kozen. Results on the propositional $\mu-$calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[73] S. A. Kripke. Semantical considerations in modal logic. *Acta Philosophica Fennica*, 16(1):83–94, 1963.

[74] O. Kupferman, P. Madhusudan, P. S. Thiagarajan, and M. Y. Vardi. Open systems in reactive environments: Control and synthesis. *Lecture Notes in Computer Science*, 1877:92+, 2000.

[75] O. Kupferman and M. Vardi. $\mu$-calculus synthesis. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer-Verlag, 2000.

[76] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In R. De Nicola, editor, *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[77] K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, Dec. 1991.

[78] X. Leroy, D. R. Damien Doligez, Jacques Garrigue, and J. Vouillon. The objective caml systemrelease 3.09, 2004.

[79] F. Levi. *Verification of temporal and Real-Time Properties of Statecharts*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1997.

[80] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag.

[81] G. Lowe. Semantic models for information flow. *Theor. Comput. Sci.*, 315(1):209–256, 2004.

[82] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (extended abstract).

[83] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proc. Concurrency Theory (CONCUR'2003), Marseille, France*, volume 2761 of *Lect.Notes Comp. Sci*, pages 251–265. Springer, Aug. 2003.

[84] N. Markey and Ph. Schnoebelen. Mu-calculus path checking. *Information Processing Letters*, 97(6):225–230, Mar. 2006.

[85] F. Martinelli. *Formal Methods for the Analysis of Open Systems with Applications to Security Properties*. PhD thesis, University of Siena, Dec. 1998.

[86] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1998.

[87] F. Martinelli. Towards automatic synthesis of systems without informations leaks. In *Proceedings of Workshop in Issues in Theory of Security (WITS)*, 2000.

[88] F. Martinelli. Module checking through patrial model checking. Technical Report IIT-TR06/2002, IIT-CNR, 2002.

[89] F. Martinelli. Analysis of security protocols as open systems. *Theoretical Computer Science*, 290(1):1057–1106, 2003.

[90] F. Martinelli and I. Matteucci. Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties, 2005. Presented at the International Workshop on Foundations of Computer Security (FCS05) - Informal proceedings.

[91] F. Martinelli and I. Matteucci. Modeling security automata with process algebras and related results, March 2006. Presented at the 6th International Workshop on Issues in the Theory of Security (WITS '06) - Informal proceedings.

[92] F. Martinelli and I. Matteucci. An approach for the specification, verification and synthesis of secure systems. *Electr. Notes Theor. Comput. Sci.*, 168:29–43, 2007.

[93] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electr. Notes Theor. Comput. Sci.*, 179:31–46, 2007.

[94] F. Martinelli and I. Matteucci. Synthesis of local controller programs for enforcing global security properties. In *Proc. First International Workshop on Advances in Policy Enforcement (APE'08)*. IEEE Computer Society Press, 2008. To appear.

[95] F. Martinelli and I. Matteucci. Synthesis of web services orchestrators in a timed setting. In *Proc. of 4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937/2008 of *Lecture Notes in Computer Science*, pages 124–138. Springer, April 13, 2008.

[96] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. *ICAS-ICNS*, 0:82, 2005.

[97] I. Matteucci. A tool for the synthesis of controller programs. In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, and S. A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 4691 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2006.

[98] I. Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electr. Notes Theor. Comput. Sci.*, 186:101–120, 2007.

[99] P. Merlin and G. V. Bochmann. On the construction of submodule specification and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5:1–25, 1983.

[100] R. Milner. Synthesis of communicating behaviour. In *Proceedings of 7th MFCS*, Poland, 1978. LNCS 64.

[101] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[102] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, N.Y., 1990.

[103] R. Milner. *Communicating and mobile systems: the $\pi$-calculus*. Cambridge University Press, 1999.

[104] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information & Computation 100*, pages 1–77, 1992.

[105] M. Müller-Olm. Derivation of characteristic formulae. In *MFCS'98 Workshop on Concurrency*, volume 18 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science B.V., 1998.

[106] M. Müller-Olm, B. Steffen, and R. Cleaveland. On the evolution of reactive components: A process-algebraic approach. In *FASE '99: Proceedings of the Second Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 161–175, London, UK, 1999. Springer-Verlag.

[107] D. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, 1981.

[108] J. Park and R. S. Sandhu. The $\text{ucon}_{\text{abc}}$ usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[109] M. Pistore, P. Roberti, and P. Traverso. Process-level composition of executable web services: "on-the-fly" versus "once-for-all" composition. In A. Gómez-Pérez and J. Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2005.

[110] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In S. Biundo, K. L. Myers, and K. Rajan, editors, *ICAPS*, pages 2–11. AAAI, 2005.

[111] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Aarhus University, 1981.

[112] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57. IEEE Computer Society Press, 1977.

[113] J. Raclet and S. Pinchinat. The control of non-deterministic systems: a logical approach. In *Proc. 16th IFAC Word Congress*, Prague, Czech Republic, jul 2005.

[114] P. Ramadge and W. Whoman. Supervision of discrete event processes. In *Proceedings of 21th IEEE conference Decision Contr.*, volume 3, pages 1228–1229, December 1982.

[115] W. Reisig. Modeling- and analysis techniques for web services and business processes. In *In FMOODS, volume 3535 of Lecture Notes in Computer Science, pages 243–258, 2005.*, 2005.

[116] R.Focardi, R.Gorrieri, and F.Martinelli. Real-time Information Flow Analysis. *IEEE JSAC*, 2003.

[117] S. Riedweg and S. Pinchinat. Maximally permissive controllers in all contexts. In *Workshop on Discrete Event Systems*, Reims, France, sep 2004.

[118] S. Riedweg and S. Pinchinat. You can always compute maximally permissive controllers under partial observation when they exist. In *Proc. 2005 American Control Conference.*, Portland, Oregon, jun 2005.

[119] G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, 2001.

[120] K. Rudie and W. Wonham. Think globally, act locally: Decentralized supervisory control. 37(11):1692–1708, Nov. 1992.

[121] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. In *CSFW '99: Proceedings of the 1999 IEEE Computer Security Foundations Workshop*, page 214, Washington, DC, USA, 1999. IEEE Computer Society.

[122] H. Saidi. Towards automatic synthesis of security protocols. March 2002.

[123] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 43, Washington, DC, USA, 2004. IEEE Computer Society.

[124] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[125] B. Steffen and A. Ingólfsdóttir. Characteristic formulae for processes with divergence. *Information and Computation*, 110(1):149–163, 1994.

[126] C. Stirling. Bisimulation, model checking, and other games, 1997. Mathfit instructional meeting on games and computation.

[127] R. S. Street and E. A. Emerson. An automata theoretic procedure for the propositional $\mu$-calculus. *Information and Computation*, 81(3):249–264, 1989.

[128] R. S. Streett and E. A. Emerson. The propositional mu-calculus is elementary. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 465–472, London, UK, 1984. Springer-Verlag.

[129] I. Ulidowski and S. Yuen. Extending process languages with time. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, London, UK, 1997. Springer-Verlag.

[130] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, The Netherlands, 1990. Springer-Verlag.

[131] I. Walukiewicz. *A Complete Deductive System for the $\mu$-Calculus*. PhD thesis, Institute of Informatics, Warsaw University, June 1993.

[132] I. Walukiewicz. On completeness of the $\mu$-calculus. In *Proceedings 8th Annual IEEE Symp. on Logic in Computer Science, LICS'93, Montreal, Canada, 19–23 June 1993*, pages 136–146, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[133] I. Walukiewicz. A complete deductive system for the $\mu$-calculus. Technical Report RS-95-6, BRICS, Department of Computer Science, University of Aarhus, Denmark, 1995.

[134] G. Winskel. On the compositional checking of validity. In *Proceedings of CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 481–501, 1990.

[135] H. Wong-Toi and D. L. Dill. Synthesizing processes and schedulers from temporal specifications. In E. M. Clarke and R. P. Kurshan, editors, *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 1990.

# Appendix A

# Technical Proofs

## A.1 Technical proofs of Chapter 3

In this section we provide the proofs of results described in Chapter 3.

In order to prove Propositions 3.1, 3.2, 3.3, 3.4, firstly we note that in our controller operators the halt condition is not explicitly given because this occurs when there are no rules that could be applied, *i.e.*, when premises of all rules are not verified. As we have already noticed, also in security automata described in Section 3.2.1, the action $\tau$ in the stop rule of each automaton is an internal action that is not really performed. So in our proofs, without loss of validity, we can omit the stop case because, looking at the semantics of each operator, it is easy to understand that the stop rule of each automata is equivalent to the halt condition of respectively operator.

In particular we prove that each security automaton is strong bisimilar to the respective controller operator. This guarantees that they satisfy the same $\mu$-calculus formula (see [116]).

**Proposition 3.1** *Let*

$$E^q = \sum_{a \in Act} \begin{cases} a.E^{q'} & \textit{iff} \quad \delta(a,q) = q' \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of a* truncation automaton $(\mathcal{Q}, q_0, \delta)$ *is also derivable from $E^q \triangleright_T F$ and vice-versa.*

*Proof:* Let $\mathcal{R}_T$ be the following relation:

$$\mathcal{R}_T = \{((\sigma, q), E^q \triangleright_T F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \overset{\sigma}{\mapsto}\}$$

where $F \overset{\sigma}{\mapsto}$ means that $F$ will perform a sequence of actions $\sigma$. We prove that $\mathcal{R}_T$ is a strong bisimulation equivalence.

- Assume that the truncation automata performs a T-Step $(\sigma, q) \xrightarrow{a}_T (\sigma', q')$. Hence $\sigma = a; \sigma'$.

We want to prove that:

$$\exists \mathcal{P} \quad E \rhd_T F \xrightarrow{a} \mathcal{P} \quad \wedge \quad ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_T$$

For the definition of the process $E^q$ we have that $E^q \xrightarrow{a} E^{q'}$ because, referring to $\delta(a, q)$, the action $a$ it is allowed in the state $q$.

According to the definition of $\mathcal{R}_T$, we know that $F \overset{\sigma}{\mapsto}$. Hence exists $F'$ such that $F \xrightarrow{a} F' \overset{\sigma'}{\mapsto}$, since $\sigma = a; \sigma'$.

According to the semantic definition of $\rhd_T$,

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \rhd_T F \xrightarrow{a} E' \rhd_T F'}$$

if $E^q \xrightarrow{a} E^{q'}$ and $F \xrightarrow{a} F'$ then $E^q \rhd_T F \xrightarrow{a} E^{q'} \rhd_T F'$. Hence, the process $\mathcal{P}$ exists and it is $E^{q'} \rhd_T F'$.

- Assume that $E^q \rhd_T F \xrightarrow{a} E^{q'} \rhd_T F'$ and $F' \overset{\sigma'}{\mapsto}$.

  We want to prove that:

  $$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{a}_T (\sigma, q)' \quad \wedge \quad (E^{q'} \rhd_T F', (\sigma, q)') \in \mathcal{R}_T$$

  Referring to the semantics definition of $\rhd_T$ operator we have that $E^q \xrightarrow{a} E^{q'}$. This means that $\delta(a, q) = q'$. Hence $\sigma = a; \sigma'$. For the rule T-Step, $(\sigma, q) \xrightarrow{a}_T (\sigma', q')$. So the couple that we are looking for is $(\sigma', q')$.

□

**Proposition 3.2** *Let* $E^{q,\omega} =$

$$\sum_{a \in Act} \begin{cases} a.E^{q',\omega} & \textit{iff} \quad \omega(a, q) = + \textit{ and } \delta(a, q) = q' \\ -a.E^{q',\omega} & \textit{iff} \quad \omega(a, q) = - \textit{ and } \delta(a, q) = q' \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let* $F$ *be the target. Each sequence of actions that is an output of a* suppression automaton $(\mathcal{Q}, q_0, \delta, \omega)$ *is also derivable from* $E^{q,\omega} \rhd_S F$ *and vice-versa.*

*Proof:* The scheme of this proof is similar to the one used into the previous proof. Let

$$\mathcal{R}_S = \{((\sigma, q), E^{q,\omega} \rhd_S F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \overset{\sigma}{\mapsto}\}$$

be a relation. We have to prove that it is a strong bisimulation relation.

According to the definition of the transition rules of suppression automata and $\rhd_S$ operators, we can divide this proof in two cases:

- Let $((\sigma, q), E^{q,\omega} \rhd_S F)$ be in $\mathcal{R}_S$. Let us suppose that $(\sigma, q) \xrightarrow{a}_S (\sigma', q')$ then $\sigma = a; \sigma'$ and $E^{q,\omega} \xrightarrow{a} E^{q',\omega}$. We want to prove that:

$$\exists \mathcal{P} \quad E^{q,\omega} \rhd_S F \xrightarrow{a} \mathcal{P} \ \wedge \ ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_S$$

  According to the definition of $\mathcal{R}_S$, exists $F'$ such that $F \xrightarrow{a} F' \overset{\sigma'}{\mapsto}$. Hence, for the first rule of $\rhd_S$ and by definition of $E^{q,\omega}$, using a similar reasoning of the proof of Proposition 3.1, we have the thesis by taking $\mathcal{P} \doteq E^{q',\omega} \rhd_S F'$.

  Let $(E^{q,\omega} \rhd_S F, (\sigma, q))$ be in $\mathcal{R}_S$ and $E^{q,\omega} \rhd_S F \xrightarrow{a} E^{q',\omega} \rhd_S F'$. We should prove that there exists a $(\sigma, q)'$ such that $(\sigma, q) \xrightarrow{a}_S (\sigma, q)'$ and $(E^{q',\omega} \rhd_S F', (\sigma, q)') \in \mathcal{R}_S$. For the rule S-StepA we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to one made in the previous proof.

- Let $((\sigma, q), E^{q,\omega} \rhd_S F)$ be in $\mathcal{R}_S$ and $(\sigma, q) \xrightarrow{\tau}_S (\sigma', q')$. We have to prove that

$$\exists \mathcal{P} \quad E^{q,\omega} \rhd_S F \xrightarrow{\tau} \mathcal{P} \ \wedge \ ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_S$$

  According to the definition of $E^{q,\omega}$ we have that $E^{q,\omega} \xrightarrow{-a} E^{q',\omega}$. Since $\sigma = a; \sigma'$ then $F \xrightarrow{a} F' \overset{\sigma'}{\mapsto}$. Hence $\mathcal{P} \doteq E^{q',\omega} \rhd_S F'$ and $((\sigma', q'), E^{q',\omega} \rhd_S F') \in \mathcal{R}_S$.

  Now assume that $(E^{q,\omega} \rhd_S F, (\sigma, q))$ be in $\mathcal{R}_S$ and $E^{q,\omega} \rhd_S F \xrightarrow{\tau} E^{q',\omega} \rhd_S F'$. We prove that

$$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{\tau}_S (\sigma, q)' \ \wedge \ (E^{q',\omega} \rhd_S F', (\sigma, q)') \in \mathcal{R}_S$$

  For the rule S-StepS we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the previous one. As a matter of fact, looking at the semantics definition of the suppression automaton we have two possible transitions labeled by $\tau$. However, we require that $(\sigma, q)'$ also satisfies that $(E^{q',\omega} \rhd_S F', (\sigma, q)') \in \mathcal{R}_S$. Hence, since $E$ changes state and goes in the state $q'$, we chose $(\sigma', q')$ as solution.

□

**Proposition 3.3** *Let* $E^{q,\gamma} =$

$$\sum_{a \in Act} \begin{cases} a.E^{q',\gamma} & \textit{iff} \quad \delta(a, q) \\ +a.b.E^{q',\gamma} & \textit{iff} \quad \gamma(a, q) = (b, q') \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of an* insertion automaton *$(\mathcal{Q}, q_0, \delta, \gamma)$ is also derivable from $E^{q,\gamma} \rhd_I F$ and vice-versa.*

*Proof:* Let $\mathcal{R}_I$ be the following relation:

$$\mathcal{R}_I = \{((\sigma, q), E^{q,\gamma} \rhd_I F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \overset{\sigma}{\mapsto}\}$$

We want to prove that $\mathcal{R}_I$ is a strong bisimulation. We have two cases:

- The first case is similar of Proposition 3.1. As a matter of fact, let $((\sigma, q), E^{q,\gamma} \rhd_I F)$ be in $\mathcal{R}_I$ and $(\sigma, q) \xrightarrow{a}_I (\sigma', q')$. We have to prove that

$$\exists \mathcal{P} \quad E^{q,\gamma} \rhd_I F \xrightarrow{a} \mathcal{P} \ \wedge \ ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_I$$

  By the first rule of $\rhd_I$ and by definition of $E^{q,\gamma}$,using a similar reasoning of the proof of Proposition 3.1, we have the thesis by taking $\mathcal{P} \doteq E^{q',\gamma} \rhd_I F'$.

  Let $(E^{q,\gamma} \rhd_I F, (\sigma, q))$ be in $\mathcal{R}_I$ and $E^{q,\gamma} \rhd_I F \xrightarrow{a} E^{q',\gamma} \rhd_I F'$. We should prove that there exists a $(\sigma, q)'$ such that $(\sigma, q) \xrightarrow{a}_I (\sigma, q)'$ and $(E^{q',\gamma} \rhd_I F', (\sigma, q)') \in \mathcal{R}_I$. For the rule I-Step we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the previous one.

- Let $((\sigma, q), E^{q,\gamma} \rhd_I F)$ be in $\mathcal{R}_I$ and $(\sigma, q) \xrightarrow{b}_I (\sigma, q')$. We have to prove that:

$$\exists \mathcal{P} \quad E^{q,\gamma} \rhd_I F \xrightarrow{b} \mathcal{P} \ \wedge \ ((\sigma, q'), \mathcal{P}) \in \mathcal{R}_I$$

  For the definition of $E^{q,\gamma}$, we have that $E^{q,\gamma} \xrightarrow{+a.b} E^{q',\gamma}$. According to the definition of $\mathcal{R}_I$, $F \overset{\sigma}{\mapsto}$, hence exists $F \overset{a;\sigma'}{\mapsto}$. Then, for the second rule of $\rhd_I$, $E^{q,\gamma} \rhd_I F \xrightarrow{b} E^{q',\gamma} \rhd_I F$. So $\mathcal{P}$ is $E^{q',\gamma} \rhd_I F$ and $((\sigma, q'), E^{q',\gamma} \rhd_I F) \in \mathcal{R}_I$.

  Now, let $(E^{q,\gamma} \rhd_I F, (\sigma, q))$ be in $\mathcal{R}_I$ and $E^{q,\gamma} \rhd_I F \xrightarrow{b} E^{q',\gamma} \rhd_I F$. We prove that

$$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{b}_I (\sigma, q)' \ \wedge \ (E^{q',\gamma} \rhd_I F, (\sigma, q)') \in \mathcal{R}_I$$

  For the rule I-Ins we have that $(\sigma, q')$ is the solution we are looking for. The reasoning is similar to the previous one.

□

**Proposition 3.4** *Let*

$$E^{q,\gamma,\omega} = \sum_{a \in Act} \begin{cases} a.E^{q',\gamma,\omega} & \textit{iff} \quad \delta(a,q) = q' \textit{ and } \omega(a,q) = + \\ -a.E^{q',\gamma,\omega} & \textit{iff} \quad \delta(a,q) = q' \textit{ and } \omega(a,q) = - \\ +a.b.E^{q',\gamma,\omega} & \textit{iff} \quad \gamma(a,q) = (b,q') \\ \mathbf{0} & \textit{othw} \end{cases}$$

*be a controller program and let $F$ be the target. Each sequence of actions that is an output of an* edit automaton $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$ *is also derivable from $E^{q,\gamma,\omega} \rhd_E F$ and vice-versa.*

*Proof:* In order to prove this lemma, we give the relation of bisimulation $\mathcal{R}_E$ which exists between edit automata and the controller operator $\rhd_E$ as follows:

$$\mathcal{R}_E = \{((\sigma, q), E^{q,\gamma,\omega} \rhd_E F) : (\sigma, q) \in \overrightarrow{Act} \times Q, E^{q,\gamma,\omega} \rhd_E F \in \mathcal{P}, F \overset{\sigma}{\mapsto}\}$$

We have to prove that it is a strong bisimulation.

We have three cases and their proofs following the reasoning made in the proofs of Proposition 3.2 and Proposition 3.3.

- Let $((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F)$ be in $\mathcal{R}_E$ and $(\sigma, q) \xrightarrow{a}_E (\sigma', q')$. We have to prove that:

$$\exists \mathcal{P} \quad E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{a} \mathcal{P} \ \wedge \ ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_E$$

  In this case, $E^{q,\gamma,\omega} \xrightarrow{a} E^{q',\gamma,\omega}$. Moreover $F \xrightarrow{a} F' \overset{\sigma'}{\mapsto}$ then $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{a} E^{q',\gamma,\omega} \triangleright_E F'$. Thus $\mathcal{P}$ is $E^{q',\gamma,\omega} \triangleright_E F'$ and $((\sigma', q'), E^{q',\gamma,\omega} \triangleright_E F') \in \mathcal{R}_E$.

  Let $(E^{q,\gamma,\omega} \triangleright_E F, (\sigma, q))$ be in $\mathcal{R}_E$ and $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{a} E^{q',\gamma,\omega} \triangleright_E F'$. We have to prove that:

$$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{a}_E (\sigma, q)' \ \wedge \ (E^{q',\gamma,\omega} \triangleright_E F', (\sigma, q)') \in \mathcal{R}_E$$

  For the rule E-StepA we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the one made in the previous proof.

- Let $((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F)$ be in $\mathcal{R}_E$ and $(\sigma, q) \xrightarrow{\tau}_E (\sigma', q')$. We have to prove that:

$$\exists \mathcal{P} \quad E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{\tau} \mathcal{P} \ \wedge \ ((\sigma', q'), \mathcal{P}) \in \mathcal{R}_E$$

  We have that $E^{q,\gamma,\omega} \xrightarrow{-a} E^{q',\gamma,\omega}$. Moreover $F \xrightarrow{a} F' \overset{\sigma'}{\mapsto}$ then $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{\tau} E^{q',\gamma,\omega} \triangleright_E F'$. Thus $(E^{q,\gamma,\omega} \triangleright_E F)'$ is $E^{q',\gamma,\omega} \triangleright_E F'$ and $((\sigma', q'), E^{q',\gamma,\omega} \triangleright_E F') \in \mathcal{R}_E$.

  Let $(E^{q,\gamma,\omega} \triangleright_E F, (\sigma, q))$ be in $\mathcal{R}_E$ and $E^{q,\omega} \triangleright_E F \xrightarrow{\tau} E^{q',\gamma,\omega} \triangleright_E F'$. We prove that

$$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{\tau}_E (\sigma, q)' \ \wedge \ (E^{q,\gamma,\omega} \triangleright_E F', (\sigma, q)') \in \mathcal{R}_E$$

  For the rule E-StepS we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the previous one.

- Let $((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F)$ be in $\mathcal{R}_E$ and $(\sigma, q) \xrightarrow{b}_E (\sigma, q')$. We prove that

$$\exists \mathcal{P} \quad E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{b} \mathcal{P} \ \wedge \ ((\sigma, q'), \mathcal{P}) \in \mathcal{R}_E$$

  According to the third rule of $\triangleright_E$, if $E^{q,\gamma,\omega}$ cannot performs the action $a$ to go into the state$E^{q',\gamma,\omega}$, *i.e.*, the action $a$ is not allowed to be performed in the state $q$ according to the definition of the function $\gamma$, and $E^{q,\gamma,\omega} \xrightarrow{+a.b} E^{q',\gamma,\omega}$, $F \xrightarrow{a} F'$ then $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{b} E^{q',\gamma,\omega} \triangleright_E F$. So $(E^{q,\gamma,\omega} \triangleright_E F)'$ is $E^{q',\gamma,\omega} \triangleright_E F$ and $((\sigma, q'), E^{q',\gamma,\omega} \triangleright_E F) \in \mathcal{R}_E$.

  Let $(E^{q,\gamma,\omega} \triangleright_E F, (\sigma, q))$ be in $\mathcal{R}_E$ and $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{b} E^{q',\gamma,\omega} \triangleright_E F$. We prove that

$$\exists (\sigma, q)' \quad (\sigma, q) \xrightarrow{b} (\sigma, q)' \ \wedge \ (E^{q',\gamma} \triangleright_E F, (\sigma, q)') \in \mathcal{R}_E$$

  For the rule E-Ins we have that $(\sigma, q')$ is the solution we are looking for. The reasoning is similar to the previous one.

□

## A.2 Technical proofs of Chapter 4

In this section we provide the proofs of results described in Chapter 4.

**Proposition 4.2** *For every* $\mathbf{K} \in \{T, S, I, E\}$ $Y \triangleright_{\mathbf{K}} X \preceq Y[f_{\mathbf{K}}]$ *holds, where* $f_{\mathbf{K}}$ *is a relabeling function depending on* $\mathbf{K}$*. In particular,* $f_T$ *is the identity function on* $Act$ *and*

$$f_S(a) = \begin{cases} \tau & \text{if } a = -a \\ a & \text{othw} \end{cases} \qquad f_I(a) = \begin{cases} \tau & \text{if } a = +a \\ a & \text{othw} \end{cases}$$

$$f_E(a) = \begin{cases} \tau & \text{if } a \in \{+a, -a\} \\ a & \text{othw} \end{cases}$$

In order to prove this proposition we prove the following four lemmas. The proof of the proposition comes trivially from the proofs of the lemmas.

**Lemma A.1** *The following relation holds*

$$Y \triangleright_T X \preceq Y[f_T] \tag{A.1}$$

*where* $f_T$ *is the identity function.*

*Proof:* We prove that the following relation is a weak simulation:

$$\mathcal{S}_T = \{(E \triangleright_T F, E[f_T]) | E, F \in \mathcal{E}\}$$

Note that being $f_T$ the identity function we could omit it without loss of generality.

Assume that $E \triangleright_T F \xrightarrow{a} E' \triangleright_T F'$ with the additional hypothesis that $F \xrightarrow{a} F'$ then, by the rule of $\triangleright_T$ we have that $E \stackrel{a}{\Rightarrow} E'$ and $(E' \triangleright_T F', E') \in \mathcal{S}_T$ according to the semantics definition of $\triangleright_T$.

It is not difficult to note that, following a similar reasoning it is also possible to prove that $Y \triangleright_T X \preceq X$.

□

**Lemma A.2** *The following relation holds*

$$Y \triangleright_S X \preceq Y[f_S] \tag{A.2}$$

*where* $f_S$ *is the same of Proposition 4.2.*

*Proof:* We prove that the following relation is a weak simulation:

$$\mathcal{S}_S = \{(E \triangleright_S F, E[f_S]) | E, F \in \mathcal{E}\}$$

There are two possible cases: The first one is when $E \triangleright_S F$ performs the action $a$. The proof of this case is the same of the proof of Lemma A.1. The second case occurs when $E \triangleright_S F \xrightarrow{\tau} E' \triangleright_S F'$. This means that $E \xrightarrow{-a} E'$ and $F$ performs an action $a$ that can not be visible from an external observer. Applying the relabeling function $f_S$ to $E$ we obtain $E_1 = E[f_S]$ such that $E_1 \xrightarrow{\tau} E_1'$, where $E_1'$ is $E'[f_S]$ and $(E' \triangleright_S F', E_1') \in \mathcal{S}_S$ for semantics definition of $\triangleright_S$.

□

**Lemma A.3** *The following relation holds*

$$Y \triangleright_I X \preceq Y[f_I] \tag{A.3}$$

*where $f_I$ is the same of Proposition 4.2.*

*Proof:* We prove that the following relation is a weak simulation:

$$\mathcal{S}_I = \{(E \triangleright_I F, E[f_I]) | E, F \in \mathcal{E}\}$$

There are two possible cases: The first one is when $E \triangleright_I F$ performs the action $a$. The proof of this case is the same of the proof of Lemma A.1. The second case occurs when $E \triangleright_I F \xrightarrow{b} E' \triangleright_I F$ means that $E \xrightarrow{+a.b} E'$ and $F$ performs an action $a$ that $E$ should not perform in order to go in the state $E'$. Applying the relabeling function $f_I$ to $E$ we obtain $E_1 = E[f_I]$ such that $E_1 \stackrel{b}{\Longrightarrow} E_1'$. where $E_1'$ is $E'[f_I]$ and $(E' \triangleright_S F', E_1') \in \mathcal{S}_I$ for semantics definition of $\triangleright_I$.

□

**Lemma A.4** *The following relation holds*

$$Y \triangleright_E X \preceq Y[f_E] \tag{A.4}$$

*where $f_E$ is the same of Proposition 4.2.*

*Proof:* We prove that the following relation is a weak simulation:

$$\mathcal{S}_E = \{(E \triangleright_E F, E[f_E]) | E, F \in \mathcal{E}\}$$

There are three possible cases: The first one occurs when $E \triangleright_E F$ performs the action $a$. The proof of this case is the same of the proof of Lemma A.1. The other two cases are the following:

- if $E \triangleright_E F \xrightarrow{\tau} E' \triangleright_E F'$ then we want to find a $E'[f_E]$ such that $E[f_E] \xrightarrow{\tau} E[f_E]'$. Referring to the second rule of the edit automata $E \triangleright_E F \xrightarrow{\tau} E' \triangleright_E F'$ when $E \xrightarrow{-a} E'$. Through the relabeling function $f_E$ we have $E[f_E] \xrightarrow{\tau} E'[f_E]$ and $(E' \triangleright_E F', E'[f_E]) \in \mathcal{S}_E$ as in proof of the Lemma A.2.

- If $E \triangleright_E F \xrightarrow{b} E' \triangleright_E F$ then we want to find a $E'[f_E]$ such that $E[f_E] \stackrel{b}{\Longrightarrow} E[f_E]'$. Referring to the last rule of edit automata $E \triangleright_E F \xrightarrow{b} E' \triangleright_E F$ when $E \xrightarrow{+a.b} E'$. Through the relabeling function $f_E$ we have $E[f_E] \stackrel{b}{\Longrightarrow} E'[f_E]$ and $(E' \triangleright_E F, E'[f_E]) \in \mathcal{S}_E$ as in proof of the Lemma A.3.

□

**Lemma 4.1**: *Let $\phi \in \forall_\wedge \mu C$ and $\psi = X$ where $X =_\nu \bigwedge_{a \in Act}([a]\mathbf{F} \vee (\langle a \rangle X \wedge [a]X))$. If $\phi$ is satisfiable then $\phi \wedge \psi$ is satisfiable.*

*Proof*: The formula $X =_\nu \bigwedge_{a \in Act}([a]\mathbf{F} \vee (\langle a \rangle X \wedge [a]X))$, or its equivalent formulation in modal $\mu-$calculus $\nu X. \bigwedge_{a \in Act}([a]\mathbf{F} \vee (\langle \alpha \rangle X \wedge [a]X))$, holds in every state (*i.e.*, it is a tautology). As a matter of fact, by this formula we are able to consider actions that are performed, *i.e.*, it is satisfied $X =_\nu \langle a \rangle X \wedge [a]X$ as well as actions that are not performed, *i.e.*, it is satisfied $[a]\mathbf{F}$. Hence let $E$ be a model of $\phi$ then $E$ is also a model for $\phi \wedge \psi$.

□

**Proposition 4.5**: *Given a formula $\phi \in \forall_\wedge \mu C$, a maximal deterministic model $E$ of this formula exists.*

In order to prove this proposition we firstly prove the following lemma.

**Lemma 4.2**: *Let $E' \models \phi$ with $\phi \in \forall_\wedge \mu C$. Let $E$ be the canonical structure of $\phi \wedge \psi$, then the following relation holds:*

$$E' \preceq E$$

*Proof*: We define the following relation:

$$\mathcal{R} = \{(E', E) | \exists \phi, E' \models \phi \in \forall_\wedge \mu C \text{ and } E \models \phi \wedge \psi$$

$$\text{and } E \text{ is the canonical structure for } \phi \wedge \psi\}$$

and prove that $\mathcal{R}$ is a simulation.

Let us suppose that $E' \xrightarrow{a} E'_1$. We want to know if exists $E_1$ such that $E \stackrel{a}{\Longrightarrow} E_1$ and $(E'_1, E_1) \in \mathcal{R}$. Since $E' \models \phi$ and $E'$ performs an action $a$ then $E' \models \langle a \rangle \varphi$ where $\varphi$ is a generic $\mu$-calculus formula. Hence $\phi \wedge \langle a \rangle \varphi \not\equiv \mathbf{F}$. According to the definition of $\psi = X$, $X =_\nu \bigwedge_{a \in Act \setminus \{\tau\}}([\alpha]\mathbf{F} \vee (\langle a \rangle X \wedge [a]X))$, it is not difficult to note that $E$, being a model for $\phi \wedge \psi$, performs the action $a$ and goes in a process $E'$ that trivially satisfies that $(E'_1, E_1) \in \mathcal{R}$ by considering $\mathbf{T}$ as formula.

□

*Proof of proposition 4.5*: It is necessary to prove that such $E$ is a model for $\phi$, that it is a maximal model and that it is a deterministic process. By Lemma 4.1 it follows that $E$ is a model for $\phi$. Being $E$ the canonical structure, it is easy to note that it is deterministic because it performs only one action $\langle \_ \rangle$ and so every rule that permits to construct it has only a premise (see rule *all*). The maximality follows from Lemma 4.2.

□

**Proposition 4.6** *If both $E$ and $F$ are weakly time alive, also $E \rhd_{\mathbf{K}} F$ is weakly time alive.*

In order to prove this proposition we prove four lemmas, one for each of the four operators. We remaind that we are working on the additional assumption that $E$ and $F$ do not perform action $\tau$.

**Lemma A.5** *If both $E$ and $F$ are weakly time alive, also $E \rhd_T F$ is weakly time alive.*

*Proof*: We want to prove that for all $(E \rhd_T F)' \in Der(E \rhd_T F)$, $(E \rhd_T F)' \stackrel{tick}{\Longrightarrow}$. Since $E$ and $F$ are weakly time alive we have:

- for all $E' \in Der(E)$ $E' \stackrel{tick}{\Longrightarrow}$
- for all $F' \in Der(F)$ $F' \stackrel{tick}{\Longrightarrow}$

Because of we are working under the additional assumption that $E$ and $F$ do not perform $\tau$ action, the transition $\stackrel{tick}{\Longrightarrow}$ can be reduced to a single step transition $\stackrel{tick}{\longrightarrow}$. Hence, from the fact that both $E$ and $F$ are weakly time alive, we know that $\exists E', F'$ that perform $\stackrel{tick}{\longrightarrow}$. According to the semantics definition of $\rhd_T$, $(E \rhd_T F)' = E' \rhd_T F' \stackrel{tick}{\Longrightarrow}$

□

**Lemma A.6** *If both $E$ and $F$ are weakly time alive, also $E \rhd_S F$ is weakly time alive.*

*Proof*: In this case the prove is very similar to the previous one, so we omit it.

□

**Lemma A.7** *If both $E$ and $F$ are weakly time alive, also $E \rhd_I F$ is weakly time alive.*

*Proof*: The proof in this case is just a bit different. We want to prove that for all $(E \rhd_I F)' \in Der(E \rhd_I F)$, $(E \rhd_I F)' \stackrel{tick}{\Longrightarrow}$. Since $E$ and $F$ are weakly time alive we have:

- for all $E' \in Der(E)$ $E' \stackrel{tick}{\Longrightarrow}$
- for all $F' \in Der(F)$ $F' \stackrel{tick}{\Longrightarrow}$

Because of we are working under the additional assumption that $E$ and $F$ do not perform $\tau$ action, the transition $\stackrel{tick}{\Longrightarrow}$ can be reduced to a single step transition $\stackrel{tick}{\longrightarrow}$. If both processes perform the $tick$ action then $(E \triangleright_I F)' = E' \triangleright_I F'$. If the $tick$ action is inserted by $E$ in the transition, $F$ does not perform any action. Since $F \in Der(F)$, we have $(E \triangleright_I F)' = E' \triangleright_I F$ and, referring to the second semantics rule of $\triangleright_I$, $E' \triangleright_I F \stackrel{tick}{\Longrightarrow}$ because also $F \stackrel{tick}{\Longrightarrow}$ since $F \in Der(F)$.

□

**Lemma A.8** *If both $E$ and $F$ are weakly time alive, also $E \triangleright_E F$ is weakly time alive.*

*Proof*: We omit the proof because it comes directly from Lemma A.6 and Lemma A.7.

□

**Proposition 4.7** *Let $E$ and $F$ be two processes and $\phi \in Fr_\mu$. If $F \preceq_t E$ then $E \models \phi \Rightarrow F \models \phi$.*

*Proof*: The proofs is similar to the proof of the result in [29] for the un-timed setting because of, referring to the Definition 2.25, by introducing the $tick$ action, the definition of weak simulation does not change.

□

**Proposition 4.8** *For every $\mathbf{K} \in \{T, S, I, E\}$ $Y \triangleright_{\mathbf{K}} X \preceq_t Y[f_{\mathbf{K}}]$ holds, where $f_{\mathbf{K}}$ is a relabeling function depending on $\mathbf{K}$. In particular, $f_T$ is the identity function on $Act$[1] and*

$$f_S(\alpha) = \begin{cases} \tau & \text{if } \alpha = -\alpha \\ \alpha & \text{othw} \end{cases} \qquad f_I(\alpha) = \begin{cases} \tau & \text{if } \alpha = +\alpha \\ \alpha & \text{othw} \end{cases}$$

$$f_E(\alpha) = \begin{cases} \tau & \text{if } a \in \{+\alpha, -\alpha\} \\ \alpha & \text{othw} \end{cases}$$

*Proof*: The proof is similar to the one of the Proposition 4.2 in Appendix A.2. As a matter of fact we have to consider the simulation condition on $tick$ action is verified. As we can see from the definition of relabeling function, the $tick$ action is consider as the other action in $Act$. It does not influence the definition of relabeling functions.

□

**Proposition 4.9** *The operators $\triangleright'$ and $\triangleright''$ enjoy Assumption 4.3.*

*Proof*: We prove the proposition for the operator $\triangleright'$. With the same argument we can prove that the proposition holds also for $\triangleright''$ operator.

We show that the following relation is a strong bisimulation:

$$\mathcal{R} = \{(E \triangleright' F, E) \mid E, F \in \mathcal{E}\}$$

[1] Here the set $Act$ must be consider enriched by control actions.

$(E \rhd' F, E) \in \mathcal{R}$ : Assume that $(E \rhd' F, E) \in \mathcal{R}$ and $(E \rhd' F, E) \xrightarrow{a} (E \rhd' F, E)'$. According to given semantics rules, $(E \rhd' F)'$ can be $E' \rhd' F$ or $E' \rhd' F'$. For both of this cases, we have that there exists $E'$ such that $E \xrightarrow{a} E'$. We have also $(E' \rhd' F, E') \in \mathcal{R}$ or $(E' \rhd' F', E') \in \mathcal{R}$. It's depend on which semantics rule of $\rhd'$ we have applied.

$(E, E \rhd' F) \in \mathcal{R}$ : Assume that is true the converse of the relation $\mathcal{R}$ and we have $E \xrightarrow{a} E'$. Using one of the two rules that we have for the monitoring operator $\rhd'$, we can have two different options for $(E \rhd' F)'$. In both cases exists $(E \rhd' F)'$ such that $(E \rhd' F) \xrightarrow{a} (E \rhd' F)'$ and $(E', (E \rhd' F)') \in \mathcal{R}$.

□

**Proposition 4.10** *Let $E$ and $F$ be two finite-state processes. If both $E$ and $F$ are weakly time alive, also $E \rhd' F$ and $E \rhd'' F$ are weakly time alive.*

*Proof*: We want to prove that for all $(E \rhd' F)' \in Der(E \rhd F)$ $(E \rhd' F)' \overset{tick}{\Rightarrow}$. $E$ and $F$ are weakly time alive so

- for all $E' \in Der(E)$ $E' \overset{tick}{\Rightarrow}$, *i.e.*, $E' \xrightarrow{\tau*} E_1 \xrightarrow{tick} E'' \xrightarrow{\tau*}$
- for all $F' \in Der(F)$ $F' \overset{tick}{\Rightarrow}$, *i.e.*, $F' \xrightarrow{\tau*} F_1 \xrightarrow{tick} F'' \xrightarrow{\tau*}$

So $\exists E', F'$ such that $(E \rhd' F)' = E' \rhd' F'$ and, referring to the semantics rules of $\rhd'$ $E' \rhd' F' \xrightarrow{\tau*} E' \rhd' F_1 \xrightarrow{\tau} E_1 \rhd' F_1 \xrightarrow{tick} E'' \rhd' F'' \xrightarrow{\tau*}$.

For the $\rhd''$ operators the proof is similar.

□

**Proposition 4.11** *The operator $\rhd'$ and $\rhd''$, that works in a timed setting, enjoy Assumption 4.4.*

*Proof* : We start by proving the proposition for $\rhd'$. In particular we have to prove the following sentence:

$$\text{For every } E \text{ and } F \text{ we have } E \rhd' F \approx_t E.$$

In order to do that, we show that the following relation is a timed weak bisimulation:

$$\mathcal{R} = \{(E \rhd' F, E) \mid E, F \in \mathcal{E} \text{ and } F \text{ is weakly time alive}\}$$

We distinguish between action $a$ and $tick$.

- Assume that $(E \rhd' F, E) \in \mathcal{R}$ and $(E \rhd' F) \xrightarrow{a} E' \rhd' F'$. According to the first semantics rules of $\rhd'$, there exists $E \stackrel{a}{\Longrightarrow} E'$.

  Assume that $(E \rhd' F, E) \in \mathcal{R}$ and $(E \rhd' F) \xrightarrow{a} E' \rhd' F$. In this cases the second rule is applied and also in this case in premises there is $E \stackrel{a}{\Longrightarrow} E'$.

  Assume that $(E \rhd' F, E) \in \mathcal{R}$ and $(E \rhd' F) \xrightarrow{a} E \rhd' F'$. In this case, $F$ performs the action $\tau$ and the third rule is applied. For the the reflexive and transitive closure of $\xrightarrow{\tau}$, we can consider $E' = E$ by the action $\tau$.

  On the other hand, let us consider $E \xrightarrow{a} E'$. Using the first or the second rule of $\rhd'$, we have two different options for $(E \rhd' F)'$. In both cases there exists $(E \rhd' F)'$ such that $(E \rhd' F) \stackrel{a}{\Rightarrow} (E \rhd' F)'$ and $(E', (E \rhd' F)') \in \mathcal{R}$.

- Assume that $(E \rhd' F, E) \in \mathcal{R}$ and $(E \rhd' F) \xrightarrow{tick} E' \rhd' F'$. We can note that this transition is possible by application of the first rule, *i.e.*, $E \xrightarrow{tick} E'$ and $F \xrightarrow{tick} F'$. So we have obviously $E'$ such that $E \stackrel{tick}{\Rightarrow} E'$.

  Assume that $(E, E \rhd' F) \in \mathcal{R}$ and $E \xrightarrow{tick} E'$. We have to prove that there exists $(E \rhd' F)'$ such that $(E \rhd' F) \stackrel{tick}{\Rightarrow} (E \rhd' F)'$. Since $F$ is weakly time alive, we consider, without loss of generality, that $F \xrightarrow{tick} F'$ [2]. Applying the first rule we obtain $E' \rhd' F'$.

For the operators $\rhd''$ we have to prove the following sentence:

$$\text{For every } E \text{ and } F \text{ we have } E \rhd'' F \approx_t E.$$

In order to do that, we show that the following relation is a timed weak bisimulation:

$$\mathcal{R} = \{(E \rhd'' F, E) \mid E, F \in \mathcal{E} \text{ and } F \text{ is weakly time alive}\}$$

We do not roundly give the proof because it is follows a reasoning very similar to the one made for $\rhd'$.

□

**Lemma 4.3** *Let $\phi$ be a safety property, conjunction of $n$ safety properties,* i.e.*, $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$ where $\phi_1, \ldots \phi_n$ are safety properties. Let $Y_1, \ldots, Y_n$ be $n$ controller programs such that $\forall i$ such that $1 \leq i \leq n$ $Y_i \models \phi_i$. We have*

$$\forall X \quad Y_n \rhd_T (Y_{n-1} \rhd_T (\ldots \rhd_T (Y_2 \rhd_T (Y_1 \rhd_T X)))) \models \phi$$

*Proof*: For induction on the number of the formulas in the conjunction $n$:

[2] In fact, being $F$ weakly time alive we know that for all $F' \in Der(F)$, $F' \stackrel{tick}{\Longrightarrow}$. This means that $F$ may perform a $\tau$ action a certain number of time. This is not a problem because it is sufficient to apply the third rule of $\rhd'$ as much time as the number of $\tau$ actions performed.

$n = 1$**:** In this case $\phi = \phi_1$. Hence $Y = Y_1$ that is the controller program such that $Y \triangleright_T X \models \phi$.

$n \Rightarrow n + 1$**:** Let $\phi$ be a formula such that $\phi = \phi_1 \wedge \ldots \wedge \phi_{n+1}$ and $Y_{n+1}$ be a controller program such that for all possible $X$, $Y_{n+1} \triangleright_T X \models \phi_{n+1}$. For inductive hypothesis we know that for all possible $X$, $Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X)))) \models \phi_1 \wedge \ldots \wedge \phi_n$. We have to prove that

$$\forall X \quad Y_{n+1} \triangleright_T (Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X))))) \models \phi$$

For sake of simplicity, we denote by $Y^n$ the process $Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X))))$. We know that for all possible $X$, $Y_{n+1} \triangleright_T X \models \phi_{n+1}$, so $Y_{n+1} \triangleright_T Y^n \models \phi_{n+1}$. For Proposition 4.1 and Lemma A.9, $Y_{n+1} \triangleright_T Y^n \models \phi_1 \wedge \ldots \wedge \phi_n$. Hence, for the definition of conjunction $Y_{n+1} \triangleright_T Y^n \models \phi$.

□

**Proposition 4.12** *Let us consider the controller operator $\triangleright_T$. It is possible to find $Y_1, \ldots, Y_n$ controller programs such that. if $Y_1 \triangleright_T X \models \phi'_1, \ldots, Y_n \triangleright_T X \models \phi_n$ then $(Y_1 \triangleright_T \ldots \triangleright_T Y_n) \triangleright_T X \models \phi_1 \wedge \ldots \wedge \phi_n$.*

In order to prove the previous proposition we prove some lemmas.

**Lemma A.9** *The following relation holds*

$$Y \triangleright_T X \preceq X \tag{A.5}$$

*Proof:* We prove that the following relation is a weak simulation.

$$\mathcal{S} = \{(E \triangleright_T F, F) | E, F \in \mathcal{E}\}$$

Assume that $E \triangleright_T F \xrightarrow{a} E' \triangleright_T F'$ with the additional hypothesis that $F \xrightarrow{a} F'$ then, by the rule of $\triangleright_T$ we have that $E \stackrel{\alpha}{\Rightarrow} E'$ and, obviously, $(E' \triangleright_T F', F') \in \mathcal{S}$.

□

**Lemma A.10** *Let $\phi$, $Y_1, \ldots, Y_n$ be as in Lemma 4.3. We have that $\forall X$*

$$\begin{array}{c} Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X)))) \models \phi \\ \Downarrow \\ (Y_n \triangleright_T \ldots \triangleright_T Y_1) \triangleright_T X \models \phi \end{array}$$

*holds.*

*Proof*: For induction on the number of controller programs $n$:

$n = 1$**:** Trivial.

$n \Rightarrow n+1$**:** For hypothesis we have that

1. $\forall 1 \leq i \leq n+1, \forall X\ Y_i \triangleright_T X \models \phi_i$;
2. $$\begin{array}{c} \forall X\ Y_n \triangleright_T (Y_{n-1} \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X)))) \models \phi \\ \Downarrow \\ \forall X\ (Y_n \triangleright_T \ldots \triangleright_T Y_1) \triangleright_T X \models \phi \end{array}$$

We want to prove that

$$\begin{array}{c} \forall X\ Y_{n+1} \triangleright_T (Y_n \triangleright_T (\ldots \triangleright_T (Y_2 \triangleright_T (Y_1 \triangleright_T X)))) \models \phi \\ \Downarrow \\ \forall X\ (Y_{n+1} \triangleright_T \ldots \triangleright_T Y_1) \triangleright_T X \models \phi \end{array}$$

For sake of simplicity we denote by $Y^n_{\triangleright_T}$ the process $(Y_n \triangleright_T \ldots \triangleright_T Y_1)$. For hypothesis 1 we can consider $Y^n$ as $X$ so, $Y_{n+1} \triangleright_T Y^n_{\triangleright_T} \models \phi_{n+1}$. For Lemma 4.3 and hypothesis 2 $Y^n_{\triangleright_T} \triangleright_T Y_{n+1} \models \phi_1 \wedge \ldots \wedge \phi_n$. Since $Y^n_{\triangleright_T} \triangleright_T Y_{n+1}$ and $Y_{n+1} \triangleright_T Y^n_{\triangleright_T}$ are bisimilar so they satisfy the same formulas (see [126]). In particular $Y_{n+1} \triangleright_T Y^n_{\triangleright_T} \models \phi_1 \wedge \ldots \wedge \phi_n$. Hence $Y_{n+1} \triangleright_T Y^n_{\triangleright_T} \models \phi$. For Lemma A.9, we conclude that

$$\forall X\ (Y_{n+1} \triangleright_T \ldots \triangleright_T Y_1) \triangleright_T X \models \phi$$

□

*Proof Proposition 4.12*: It follows directly from proofs of Lemma 4.3 and Lemma A.10.

□

**Proposition 4.13** *Given the system $P_k \| X$. If $\phi$ is an invariant formula for the system $P \| X$ then*

$$\forall X \quad (\forall n \quad P_n \| X \models \phi \quad \textit{if and only if} \quad X \models \phi)$$

*Proof*: The proof comes directly from the Lemma 2.6. As a matter of fact, according to Lemma 2.6 and to Definition 4.3:

$$P_n \| X \models \phi \text{ if and only if } P_{n-1} \| X \models \phi_{//_P} \equiv \phi$$

Reiterating this procedure $n$ times we obtain:

$$\forall X \quad (\forall n \quad P_n \| X \models \phi \quad \text{if and only if} \quad X \models \phi)$$

□

**Proposition 4.14**: *Let $P$ and $Q$ be two finite state processes,*

$$Q \triangleright P \models \phi \text{ if and only if } Q \models \phi//_{\triangleright P}$$

*Proof (Sketch):* The proof of this proposition is done by induction on the complexity of the formula we consider. Here we give a sketch of the proof by proving the proposition for conjunction. Following the reasoning in [3] we obtain the entire proof.

Let $\phi = \phi_1 \wedge \phi_2$ be the considered formula. We want to prove that $Q \triangleright P \models \phi$ if and only if $Q \models \phi//_{\triangleright P}$. $Q \triangleright P \models \phi$ if and only if $Q \triangleright P \models \phi_1 \wedge \phi_2$ if and only if $Q \triangleright P \models \phi_1$ and $Q \triangleright P \models \phi_2$. For inductive hypothesis, $Q \triangleright P \models \phi_1$ if and only if $Q \models \phi_1//_{\triangleright P}$ and $Q \triangleright P \models \phi_2$ if and only if $Q \models \phi_2//_{\triangleright P}$. Hence $Q \triangleright P \models \phi$ if and only if $Q \models \phi_1//_{\triangleright P}$ and $Q \models \phi_2//_{\triangleright P}$ if and only if $Q \models \phi_1//_{\triangleright P} \wedge \phi_2//_{\triangleright P}$.

□