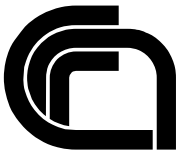


*Consiglio Nazionale delle Ricerche*

# A framework for the modeling and synthesis of security automata based on process algebras

F. Martinelli, I. Matteucci

IIT TR-04/2007

**Technical report**

**Marzo 2007**

**Istituto di Informatica e Telematica**

# A framework for the modeling and synthesis of security automata based on process algebras *

Fabio Martinelli[1], Ilaria Matteucci[1,2]
Istituto di Informatica e Telematica - C.N.R., Pisa, Italy[1]
Dipartimento di Matematica, Università degli Studi di Siena[2]
{Fabio.Martinelli, Ilaria.Matteucci}@iit.cnr.it

**Abstract**

In this paper we present our approach for the modeling and the synthesis of *enforcement mechanisms*that are mechanism able to force security policies. In particular, starting from the definition of *security automata* introduced in the literature by Schneider, Ligatti et *al.*, we define a set of process algebra operators, said *controller operators*, able to mimic the security automata's behavior. Hence we present semantics definitions of four different controller operators that act by monitoring possible un-trusted component of a given system. They guarantee that the whole system is secure, i.e. it works as prescribed by a given security policy. We also present our theory for automatically building a process that is a controller program for a chosen controller operator. By exploiting satisfiability results on temporal logic we have developed a tool that generates such processes. The tool implements the partial model checking technique and a satisfiability procedure for a modal $\mu$-calculus formula.

We then present how it is possible to extend our approach in a timed setting and to deal with parameterized systems.

**Keywords**: Partial model checking, process algebra operators, security property, controller operator, synthesis of controller program.

## 1 Introduction

In the last few years the amount or information and sensible data that circulate on the net has been growing up. This is one of important reasons that have contributed to increase research on the definition of formal method for the analysis and the verification of secure systems, i.e. systems that satisfy some security properties that specify acceptable executions of programs.

An interesting approach is based on the idea that potential attackers should be analyzed as if they were un-specified components of a system; thus reducing security

*This work is an expanded and revised version of [30, 31].

analysis to the analysis of *open systems*. As a matter of fact the behavior of an open system may be not completely specified and may present some uncertainty (see [29]).

Recently the interest on developing techniques to study how to make a system secure by enforcing *security policy* is growing (e.g. see [5, 9, 10, 41]).

We have extended the verification approach of [29] with a method for automatically enforcing the desired security property. As a matter of fact we define process algebra operators (see [32]) said *controller operators* and denoted by $Y \triangleright X$, where $Y$ is the *controller program* and $X$ is the *target system*, i.e. a possible un-trusted component.

Schneider in [41] has defined the concept of enforcement mechanism as a program that control that a given security property is respected. He has also given a definition of *security automaton* as an automaton that processes a sequence of input actions that has finite or infinite length. It works by monitoring the target system, i.e. an application whose behavior is unknown, and terminating any execution that is about to violate the security policy being enforced. Starting form his definition, Ligatti et al. described four different ways to enforce safety policies ([9, 10]). The **truncation automaton** can recognize bad sequences of actions and halts program execution before a security property is violated, but cannot otherwise modify program behavior. The **suppression automaton** can suppress individual program actions without terminating the program outright in addition to being able to halt program execution. The third automaton is the **insertion automaton** that is able to insert a sequence of actions into the program actions stream as well as terminate the program. The last one is the **edit automaton**. It combines the power of suppression and insertion automaton hence it is able to truncate actions sequences and can insert or suppress security-relevant actions at will.

In this paper, we model security automata defined in [9, 10] through process algebra by defining *controller operators* $Y \triangleright_{\mathbf{K}} X$, where $\mathbf{K} \in \{T, S, I, E\}$ where $T$, $S$, $I$ and $E$ represent Truncation, Suppression, Insertion and Edit automaton respectively. We give the semantics definition of each of controller operator and prove that they have the same behavior of the respective security automaton.

In order to express security policies we use equational $\mu$-calculus formulae because many properties of systems are naturally specified by means of fixed points and it is very expressive.

Hence, at the beginning, we have a system $S$ and an equational $\mu$-calculus formula $\phi$ that express a safety policy. Our goal is to guarantee that $\forall X$, $S \| X \models \phi$. First of all we apply the *partial model checking* function 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 necessary/untrusted part of the system, here $X$. Hence we force $X$ to enjoy $\phi'$ by using an appropriate controller $Y \triangleright_{\mathbf{K}} X$.

Our approach permits us to automatically synthesize a controller program $Y$ for a given controller operator $Y \triangleright_{\mathbf{K}} X$ by exploiting satisfiability procedure for the $\mu$-calculus. Moreover we show our tool that is effectively able to generate a controller program $Y$ starting from a system $S$ and $\phi$.

An advantage of this 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 component $X$ to enjoy a given property, by treating it as the whole system of interest. However, often not all the 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.

In the last part of the paper we present further results on how our controller operators can be use also to force security policies in a timed setting and to treat with parameterized systems, $S = P_n$ where $n$ is the parameter and $P_n = \underbrace{P \| P \| \ldots \| P}_{n}$.

Our logical approach is also able to deal with composition problems, that have been considered as an interesting issue in [9]. As a matter of fact we present how we are able to enforce a policies that is a composition of several sub-policies.

*This paper is organized as follows*. Section 2 presents some related work, Section 3 recalls basic theory about process algebras, modal logic and the partial model checking technique. Section 4 briefly explains how we use open system for security analysis. Section 5 describes our controller operators and shows how they model security automata. Section 6 presents our theory for the synthesis of process algebra controller operators and describes the architecture of our tool. Section 7 shows an example of application. Section 8 presents some related results and Section 9 concludes the paper.

# 2 Related work

In the literature a lot of works are about the study of enforceable properties and mechanism. In this paper we deal with two different aspect, the modeling of security automata and the synthesis of controller program.

Security automata was introduced by Schneider in [41] as a triple $(\mathcal{Q}, q_0, \delta)$ where $\mathcal{Q}$ is a set of states, $q_0$ is the initial state and, being $Act$ the set of security-relevant actions, $\delta : Act \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$ is the transition function. A security automaton processes a sequence of actions $a_1 a_2 \ldots$ one by one. For each action, the current global state $\mathcal{Q}'$ is calculated, by initially starting from $\{q_0\}$. As each $a_i$ is read, the security automaton changes $\mathcal{Q}'$ in $\bigcup_{q \in \mathcal{Q}'} \delta(a_i, q)$. If the automaton can make a transition on a given action, i.e. $\mathcal{Q}'$ is not empty, then the target is allowed to perform that action. The state of the automaton changes according to transition rules. Otherwise the target execution is terminated. A security property that can be enforced in this way corresponds to a *safety property* (according to [41], a property is a safety one, if whenever it does not hold in a trace then it does not hold in any extension of this trace).

Starting from the Schneider's work, Ligatti et al. in [9, 10] have defined four different kinds of security automata which deal with finite sequences of actions: **truncation automaton**, **suppression automaton**, **insertion automaton** and **edit automaton**.

Our work represents a significant contribution to the previous works (see [9, 10, 25, 41]), because by modeling these automata by process algebra operators we are able to deal also with the synthesis problem. This problem for the security automata was not addressed in previous works. In fact, most of the related works deal with the verification rather than with the synthesis problem.

Other works present different frameworks to model, analyze and study security automata, but do not deal with the synthesis problem. In [7], for example, the authors propose, by using $CSP-OZ$, a specification language combining *Communicating Se-*

*quential 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 refinement can be used to reason about specifications of security automata and their combination with target systems.

Also Bartoletti, Degano and Ferrari in [6] refer to [41] 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*. On the contrary we monitor the target at run-time.

The synthesis problem is addressed in different topic (e.g. [4, 40, 23, 48] ).

In [28], 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 [18]). That work did not deal with controllers.

On the other hand much of prior works are about the study of enforceable properties and related mechanisms but they do not deal with synthesis problem. In [16] the authors deal with a safety interface that permits to study if a module is safe or not in a given environment.

We use controller synthesis in order to force a system to guarantee security policy. The synthesis of controllers is also, however, studied in other research areas. There are approaches exploits satisfiability procedure. Usually this kind of approaches are used when properties are expressed using linear time logic or similar [19, 39]. Many approaches to the controller synthesize problem are based on game theory. As 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. Example of these paper are [3, 22, 26, 35, 36, 38].

# 3 Process algebra, logics and partial model checking

In this section we show preliminary notions that are useful to understand the results that we are going to present in this work.

## 3.1 A process algebra

In this subsection we recall the $CCS$ process algebra introduced by Milner in [33]. We describe the semantics of $CCS$ by using the *Generalized Structural Operational Semantics*, $GSOS$ for short (see [27]). This format of operational semantics was introduced by Bloom *et al.* in [1, 12, 13] by following the treatment proposed by Simpson in [42]. We choose to introduce this semantics specification because it is more suitable than $SOS$ for defining controller operators behavior.

### 3.1.1 Generalized Structural Operational Semantics

Let $V$ be a set of variables, ranged over by $x, y, \ldots$, and let $Act$ be a finite set of actions, ranged over by $a, b, c, \ldots$. A *signature* $\Sigma$ is a pair $(F, ar)$ where:

- $F$ is a set of function symbols, disjoints from $V$,
- $ar : F \mapsto \mathbb{N}$ is a *rank function* which gives the arity of a function symbol; if $f \in F$ and $ar(f) = 0$ then $f$ is called a *constant symbol*.

Given a signature, let $W \subseteq V$ be a set of variables. It is possible to define the set of $\Sigma$-*terms* over $W$ as the least set such that every element in $W$ is a term and if $f \in F$, $ar(f) = n$ and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term. It is also possible to define an *assignment* as a function $\gamma$ from the set of variables to the set of terms such that $\gamma(f(t_1, \ldots, t_n)) = f(\gamma(t_1), \ldots \gamma(t_n))$. Given a term $t$, let $Vars(t)$ be the set of variables in $t$. A term $t$ is *closed* if $Vars(t) = \emptyset$.

Now we are able to describe the *GSOS* format. A *GSOS* rule $r$ has the following format:

$$\frac{\{x_i \xrightarrow{a_{ij}} y_{ij}\}_{1 \le j \le m_i}^{1 \le i \le k} \quad \{x_i \not\xrightarrow{b_{ij}}\}_{1 \le j \le n_i}^{1 \le i \le k}}{f(x_1, \ldots, x_k) \xrightarrow{c} g(\vec{x}, \vec{y})} \tag{1}$$

where all variables are distinct; $\vec{x}$ and $\vec{y}$ are the vectors of all $x_i$ and $y_{ij}$ variables respectively; $m_i, n_i \ge 0$ and $k$ is the arity of $f$. We say that $f$ is the *operator* of the rule ($op(r) = f$) and $c$ is the action. A *GSOS* system $\mathcal{G}$ is given by a signature and a finite set of *GSOS* rules. Given a signature $\Sigma = (F, ar)$, an assignment $\zeta$ is *effective* for a term $f(s_1, \ldots, s_k)$ and a rule $r$ if:

1. $\zeta(x_i) = s_i$ for $1 \le i \le k$;
2. for all $i, j$ with $1 \le i \le k$ and $1 \le j \le m_i$, it holds that $\zeta(x_i) \xrightarrow{a_{ij}} \zeta(y_{ij})$;
3. for all $i, j$ with $1 \le i \le k$ and $1 \le j \le n_i$, it holds that $\zeta(x_i) \not\xrightarrow{b_{ij}}$,

The formal semantics of terms is described by a *labeled transition system*, $LTS$ for short. It is a pair $(\mathcal{E}, \mathcal{T})$ where $\mathcal{E}$ is the set of terms and $\mathcal{T}$ is a ternary relation $\mathcal{T} \subseteq (\mathcal{E} \times Act \times \mathcal{E})$, known as a *transition relation*. The transition relation among closed terms can be defined in the following way: $f(s_1, \ldots, s_n) \xrightarrow{c} s$ if and only if there exists an *effective* assignment $\zeta$ for a rule $r$ with operator $f$ and action $c$ such that $s = \zeta(g(\vec{x}, \vec{y}))$. There exists a unique transition relation induced by a *GSOS* system (see [13]) and this transition relation is *finitely branching*.

### 3.1.2 $CCS$ process algebra

*Process algebras* (or *process calculi*) are approaches to formally modeling concurrent systems. Process algebras provide a method for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. An interesting process calculi is the *Calculus of Communicating Systems*, $CCS$ for short, developed by Robin Milner (see [33]). Its actions model

Prefixing:

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

Choice:

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$$

Parallel:

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y} \qquad \frac{y \xrightarrow{a} y'}{x \| y \xrightarrow{a} x \| y'} \qquad \frac{x \xrightarrow{l} x' \quad y \xrightarrow{\bar{l}} y'}{x \| y \xrightarrow{\tau} x' \| y'}$$

Restriction:

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

Relabeling:

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

Table 1: *GSOS* system for $CCS$.

indivisible communications between exactly 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.

Let $\mathcal{L} \subseteq Act$ be a finite set of actions, $\bar{\mathcal{L}} = \{\bar{a} \mid a \in \mathcal{L}\}$ be the set of complementary actions where $\bar{}$ is a bijection with $\bar{\bar{a}} = a$, $Act_\tau$ be $\mathcal{L} \cup \bar{\mathcal{L}} \cup \{\tau\}$, where $\tau$ is the special action that denotes an internal computation step (or communication) and $\Pi$ be a set of constant symbols that can be used to define processes with recursion. We define the signature $\Sigma_{CCS} = (F_{CCS}, ar)$ as follows.

$$F_{CCS} = \{\mathbf{0}, +, \|\} \cup \{a.|a \in Act_\tau\} \cup \{\backslash L | L \subseteq \mathcal{L} \cup \bar{\mathcal{L}}\} \cup \\ \cup \quad \{[f] | f : Act_\tau \mapsto Act_\tau\} \cup \Pi$$

where $f(\tau) = \tau$. The function $ar$ is defined as follows: $ar(\mathbf{0}) = 0$ and for every $\pi \in \Pi$ we have $ar(\pi) = 0$, $\|$ and $+$ are binary operators and the other ones are unary operators.

The operational semantics of $CCS$ closed terms is given in Table 1 by means of the *GSOS* and by $LTS$ $(\mathcal{E}, \mathcal{T})$, where $\mathcal{E}$ is a set of process terms ranged over by $E, F, P, Q, \ldots$, and $\mathcal{T}$ is a transition relation. We denote by $Der(E)$ the set of derivatives of a (closed) term $E$, i.e. the set of processes that can be reached from $E$ through the transition relation $\mathcal{T}$.

Informally the semantics of $CCS$ terms is the following:

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

**Choice:** the term $E + F$ represents the non-deterministic choice between the processes $E$ and $F$. Choosing the action of one of the two components, the other is dropped.

**Parallel composition:** the term $E \| F$ represents the parallel composition of the two processes $E$ and $F$. It can perform an action if one of the two processes can perform an 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 the successors of each component, respectively.

**Restriction:** the process $E \backslash L$ behaves like $E$ 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.

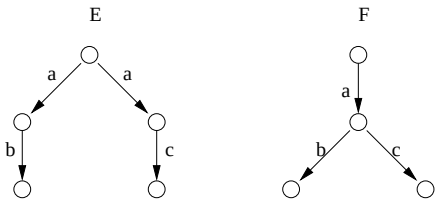
**Relabeling:** the process $E[f]$ behaves like the $E$ but the actions are renamed $via f$.

### 3.1.3 Behavioral Equivalences

There are a lot of scenarios in which it is important to understand when two different processes have the same behavior. Several behavioral relations are defined in order to compare the behavior of different processes. Here we are interested in *strong and weak simulation* and *bisimulation*

**Strong simulation and bisimulation equivalences** Look at the following example:

**Example 3.1** *Consider two vendor machine $E$ and $F$ which behaviors can be represented by the following figure:*



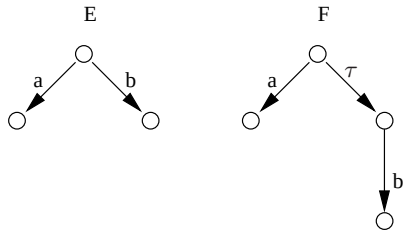

*These two process are not equivalent. To underline the way they differ, we introduce a notion of* simulation *according to which $F$ can simulate $E$, but not viceversa. Informally, to say "$F$ simulates $E$" means that $F$'s behavior pattern is at least as rich as that of $E$.*

**Definition 3.1** *Let $(\mathcal{E}, \rightarrow)$ be an $LTS$ of concurrent processes over the set of actions $Act_\tau$, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* strong simulation, *denoted by $\prec$, over $(\mathcal{E}, \rightarrow)$ if and only if, whenever $(E, F) \in \mathcal{R}$ we have:*

$$\text{if } E \xrightarrow{a} E' \text{ then } \exists\, F' \text{ s.t. } F \xrightarrow{a} F' \text{ and } (E', F') \in \mathcal{R}.$$

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

**Weak simulation and bisimulation equivalences** Look at the following figure:




The processes $E$ and $F$ cannot be consider equivalent, since the second perform an internal action by reaching a state where an action $a$ is no longer possible. To compare two processes like the processes in the previous figure Milner in [33] proposed the notion of *weak bisimulation*.

Let $\hat{\tau} = \epsilon$ and if $a \neq \tau$ then $\hat{a} = a$. Moreover, we have

$$E \stackrel{\tau}{\Longrightarrow} E' \ (E \Rightarrow E' \text{ or } E \stackrel{\epsilon}{\Longrightarrow} E') \text{ if } E \stackrel{\tau}{\rightarrow}^{*} E'$$
$$E \stackrel{\hat{a}}{\Longrightarrow} E' \qquad \text{if } E \stackrel{\tau}{\Longrightarrow} \stackrel{\hat{a}}{\longrightarrow} \stackrel{\tau}{\Longrightarrow} E'$$

where $E \stackrel{\tau}{\longrightarrow}^{*} E'$ is the transitive and reflexive closure of $\stackrel{\tau}{\longrightarrow}$. Note that $E \stackrel{\tau}{\Longrightarrow} \stackrel{\hat{a}}{\longrightarrow} \stackrel{\tau}{\Longrightarrow} E'$ is a short notation for $E \stackrel{\tau}{\Rightarrow} E_\tau \stackrel{\hat{a}}{\rightarrow} E'_\tau \stackrel{\tau}{\Rightarrow} E'$ where $E_\tau$ and $E'_\tau$ denote intermediate states that is not important for this framework.

The *weak bisimulation* relation permits to abstract to some extent from the internal behavior of the system, represented by the internal $\tau$ action.

**Definition 3.2** *Let $(\mathcal{E}, \rightarrow)$ be an $LTS$ of concurrent processes over the set of actions $Act_\tau$, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* weak simulation, *denoted by $\preceq$, over $(\mathcal{E}, \rightarrow)$ if and only if, whenever $(E, F) \in \mathcal{R}$ we have:*

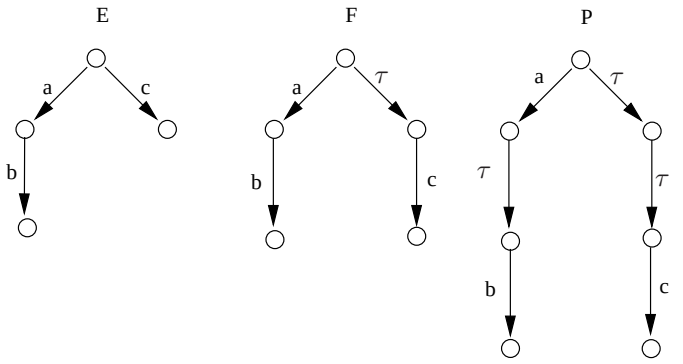
$$\text{if } E \stackrel{a}{\longrightarrow} E' \text{ then } \exists\ F' \text{ s.t. } F \stackrel{a}{\Longrightarrow} F' \text{ and } (E', F') \in \mathcal{R},$$

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

An important result proved by Milner is the following.

**Proposition 3.1 ([33])** *Every strong simulation is also a weak one.*

**Example 3.2** *Let we consider three different processes $E$, $F$ and $P$ as in the following figure. It is easy to not that $F$ and $P$ are weakly bisimilar. On the contrary $E$ and $F$ ($P$) are not weakly bisimilar.*




$E \not\approx F \approx P$

## 3.2 Two variants of $\mu$-calculus

In this subsection we describe two different variants of $\mu$-calculus: *modal $\mu$-calculus* and *equational $\mu$-calculus*.

**Modal $\mu$-calculus** Modal $\mu$-calculus is a process logic which extends $HML$ logic (see [20]) by adding fix-point operators in order to reason directly about recursive definitions of properties. It permits us to analyze non terminating behavior of systems. It is a powerful temporal logic which subsumes several other logics such as $CTL$,$CTL^*$ and $ECTL^*$ (see [11, 17, 46]). As usual for $\mu$-calculi, for the interpretation of the formulas we might consider $LTS$.

Let $a$ be in $Act_\tau$ and $Z$ be a variable ranging over a finite set of variables $V$, formulae 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 definitions of bound and free variables. The interpretation of a closed formula $\phi$ w.r.t. an $LTS$ $M$ 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) fix-point of this function. The interpretation of a formula with free variable is a monotonic function, so a least (greatest) fix-point exists.

Formally, given an $LTS$ $M = \langle S, \rightarrow \rangle$, the semantics of a formula $\phi$ is a subset $[\![\phi]\!]_\rho$ of the states of $M$, defined in Table 2, where $\rho$ is a function (called *environment*) from free variables of $\phi$ to subsets of the states of $M$. The environment $\rho[S'/Z](Y)$ is equal to $\rho(Y)$ if $Y \neq Z$, otherwise $\rho[S'/Z](Z) = S'$.

**Equational $\mu$-calculus** Equational $\mu$-calculus is based on fix-point equations instead of fixpoint operators that permit to define recursively the properties of systems. A

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

Table 2: Denotational semantics of modal $\mu$-calculus.

*minimal (maximal) fix-point equation* is $Z =_\mu \phi$ ($Z =_\nu \phi$), where $\phi$ is an assertion, i.e. a simple modal formula without recursion operators.

**Example 3.3** *A lot of properties can be defined by using equational $\mu$-calculus. In particular it is useful to express several security properties. For instance it is possible to find a formula to express safety property as, for instance, a formula that expresses the possibility to open a new file only if the previous one is closed:*

$$
\begin{array}{l}
X =_\nu [\tau]X \wedge [open]Y \\
Y =_\nu [\tau]Y \wedge [close]X \wedge [open]F
\end{array}
$$

. *A* liveness property *("something good happens") like "a state satisfying $\phi$ can be reached" is expressed by $Z =_\mu \langle\_\rangle Z \vee \phi$*[1].

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 w.r.t. 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 $M = \langle S, \rightarrow \rangle$ 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

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

environments. Let $\sigma$ be in $\{\mu, \nu\}$, $\sigma U.f(U)$ represents the $\sigma$ fix-point 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 $D$.

For both of these logics the following theorem holds.

**Theorem 3.1 ([44])** *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.*

Later in the paper we use the finitary axioms system proposed by Walukievicz in [47] in order to synthesize controller program for given controller operator (Section 6).

### 3.2.1 Characteristic formula

A *characteristic formula* (see [34]) is a formula in equational $\mu$-calculus that completely characterizes the behavior of a (state in a) state-transition graph modulo a chosen notion of behavioral relation. It is possible to define the notion of characteristic formula for a given finite state process $E$ with respect different behavioral relation. In this subsection we present the notion of characteristic formula for $E$ w.r.t. simulation and bisimulation relations.

**Definition 3.3** *Given a finite state process $E$, its characteristic formula (w.r.t. weak bisimulation) $D_E \downarrow Z_E$ is defined by the following equations for every $E' \in Der(E)$, $a \in Act$:*

$$Z_{E'} =_\nu \Big( \bigwedge_{\substack{a \in Act_\tau \\ E' \xrightarrow{a} E''}} \langle\langle \hat{a} \rangle\rangle Z_{E''} \Big) \wedge \Big( \bigwedge_{a \in Act_\tau} ([a]( \bigvee_{E' \stackrel{\hat{a}}{\Longrightarrow} E''} Z_{E''})) \Big)$$

where $\langle\langle a \rangle\rangle$ of the modality $\langle a \rangle$ which can be introduce as abbreviation (see [34]):

$$\langle\langle \epsilon \rangle\rangle \phi \stackrel{def}{=} Z \text{ where } Z =_\mu \phi \vee \langle \tau \rangle Z$$

$$\langle\langle a \rangle\rangle \phi \stackrel{def}{=} \langle\langle \epsilon \rangle\rangle \langle a \rangle \langle\langle \epsilon \rangle\rangle \phi$$

The following lemma characterizes the power of these formulae.

**Lemma 3.1** *Let $E_1$ and $E_2$ be two different finite-state processes. If $\phi_{E_2}$ is characteristic for $E_2$ then:*

1. *If $E_1 \approx E_2$ then $E_1 \models \phi_{E_2}$*

2. *If* $E_1 \models \phi_{E_2}$ *and* $E_1$ *is finite-state then* $E_1 \approx E_2$.

Now we consider weak simulation as behavioral relation and we define the characteristic formula of a finite-state process $E$ w.r.t. this relation as follows.

**Definition 3.4** *Given a finite state process* $E$*, its characteristic formula (w.r.t. weak simulation)* $D_E \downarrow Z_E$ *is defined by the following equations: for every* $E' \in Der(E)$*,*

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

Following the reasoning used in [34], the following proposition holds.

**Lemma 3.2** *Let* $E$ *be a finite-state process and let* $\phi_{E,\preceq}$ *be its characteristic formula w.r.t. simulation, then* $F \preceq E \Leftrightarrow F \models \phi_{E,\preceq}$.

It is easy to note that the characteristic formula of a process w.r.t. simulation is weaker than the formula defined in the Definition 3.3.

## 3.3 Partial model checking

The partial model checking mechanisms was introduced by Andersen in [2]. This technique relies upon compositional methods for proving properties of concurrent systems.

The intuitive idea underlying the partial model checking is the following: proving that $E_1 \| E_2$ satisfies $\phi$ is equivalent to prove that $E_2$ satisfies a modified specification $\phi_{//_{E_1}}$, where $//_{E_1}$ is the partial evaluation function for the parallel composition operator (see [2] or Table 3). Hence, the behavior of a component has been partially evaluated and the requirements are changed in order to respect this evaluation. The partial model checking function (also called partial evaluation) for the parallel operator is given in Table 3.

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

**Example 3.4** *Let* $[\tau]\phi$ *be the given formula and let* $E \| F$ *a process. We want to evaluate the formula* $[\tau]\phi$ *w.r.t. the* $\|$ *operator and the process* $E$*. The formula* $[\tau]\phi_{//_E}$ *is satisfied by* $F$ *if the following three condition hold:*

- $F$ *performs an action* $\tau$ *going in a state* $F'$ *and* $E \| F'$ *satisfies* $\phi$*; this is taken into account by the formula* $[\tau](\phi_{//_E})$*;*
- $E$ *performs an action* $\tau$ *going in a state* $E'$ *and* $E' \| F$ *satisfies* $\phi$*, and this is considered by the conjunction* $\wedge_{E \stackrel{\tau}{\longrightarrow} E'} \phi_{//_{E'}}$*, where every formula* $\phi_{//_{E'}}$ *takes into account the behavior of* $F$ *in composition with a* $\tau$ *successor of* $E$*;*
- *the* $\tau$ *action is due to the performing of two complementary actions by the two processes. So for every* $a$ *successor* $E'$ *of* $E$ *there is a formula* $[\bar{a}](\phi_{//_{E'}})$.

In [2], the following lemma is given.

Parallel:

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

Relabeling:

$$
\begin{array}{lcl}
Z//[f] & = & Z \\
(Z =_\sigma \phi D)//[f] & = & (Z =_\sigma \phi//[f](D)//[f]) \\
\langle a\rangle\phi//[f] & = & \bigvee_{b:f(b)=a}\langle b\rangle(\phi//[f]) \\
[a]\phi//[f] & = & \bigwedge_{b:f(b)=a}[b](\phi//[f]) \\
\phi_1 \wedge \phi_2//[f] & = & (\phi_1//[f]) \wedge (\phi_2//[f]) \\
\phi_1 \vee \phi_2//[f] & = & (\phi_1//[f]) \vee (\phi_2//[f]) \\
\mathbf{T}//[f] & = & \mathbf{T} \\
\mathbf{F}//[f] & = & \mathbf{F}
\end{array}
$$

Table 3: Partial evaluation function for parallel operator and relabeling operator.

**Lemma 3.3** *Given a process $E\|F$ (where $E$ is finite-state) and an equational specification $D \downarrow Z$ we have:*

$$E\|F \models (D \downarrow Z) \text{ iff } F \models (D \downarrow X)_{//_E}$$

Remarkably, this function is exploited in [2] to perform model checking efficiently, i.e. both $E$ and $F$ are specified. In our setting, the process $F$ will be not specified.

It is important to note that a lemma similar to Lemma 3.3 holds for each $CCS$ operators. As a matter of fact in Table 3 we also recall the definition of the partial model checking function for relabeling operators. For the other $CCS$ operators see [2].

# 4 Open system analysis for security analysis

In this section we recall the concept of *open system* that we use to study our systems in order to guarantee that they are secure.

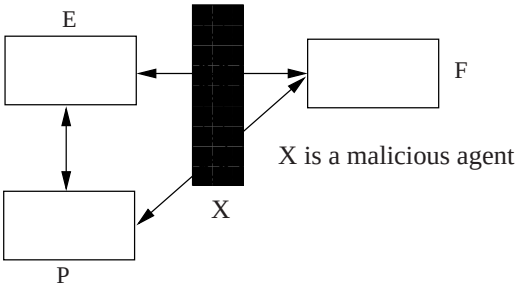
A system is *open* if it has some unspecified components. We want to make sure that the system with an unspecified component works properly, e.g. fulfills a certain

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 for the description of system behavior, 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 4.1** *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 the following figure:*




Specification : $E\|F\|P\|X$

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 the malicious users or intruders may have, which is exactly a *verification* problem of open systems. According to [27, 29], the problem that we want to study can be formalized as follows:

$$\text{For every component } X \quad S\|X \models \phi \tag{2}$$

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 component (i.e. intruder, malicious user, hostile environment, *etc.*) which may possibly interact with it.

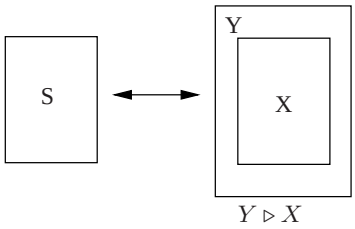
By using partial model checking we reduce such a verification problem as in Formula (2) to a validity checking problem as follows:

$$\forall X \quad S\|X \models \phi \quad \text{iff} \quad X \models \phi_{//_S} \tag{3}$$

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

# 5 Process algebra controller operators

Using the open system approach we develop a theory to enforce security properties. 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), 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.




Specification : $S \| Y \triangleright X$

Hence we use controller operator in such way the specification of the system becomes:

$$\exists Y \quad \forall X \quad \text{s.t.} \quad S \| (Y \triangleright X) \models \phi \tag{4}$$

By partially evaluating $\phi$ w.r.t. $S$ the Formula (4) is reduced as follows:

$$\exists Y \quad \forall X \quad Y \triangleright X \models \phi' \tag{5}$$

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

It is important to note that, by using partial model checking we need to control only the possible un-trusted component of the system. Our method allows one to monitor only the necessary/untrusted part of the system, here $X$. We can define several kinds of controller operators. Each of them has different capabilities. We deal with security automata (*truncation, suppression, insertion, edit*) defined in [9, 10] by modeling them by process algebra controller operators $Y \triangleright_{\mathbf{K}} X$, where $\mathbf{K} \in \{T, S, I, E\}$ where $T$ stays for *Truncation*, $S$ for *Suppression*, $I$ for *Insertion* and $E$ for *Edit*. In the next section we just recall the semantics definition of security automata and we present how we model them by process algebra operators giving the semantics of our process algebra operators.

## 5.1 Modeling security automata with process algebra

Here we choose to follow the approach given by Ligatti and al. in [9] to describe the behavior of security automata.

A *security automaton* at least consists of a (countable) set of states, say $\mathcal{Q}$, a set of actions $Act$ and a transition (partial) function $\delta$. Each kind of automata has a slightly

different sort of transition function $\delta$, and these differences account for the variations in their expressive power. The exact specification of $\delta$ is part of the definition of each kind of automaton. We use $\sigma$ to denote a sequence of actions, $\cdot$ for the empty sequence and $\tau$[2] to represent an internal action.

The execution of each different kind of security automata $\mathbf{K}$ 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 action sequence and the state after that the automaton takes a single step, and $a$ denotes the action produced by the automaton. The single-step judgment can be generalized to a multi-step judgment $(\sigma, q) \stackrel{\gamma}{\Longrightarrow}_{\mathbf{K}}$[3] $(\sigma', q')$, where $\gamma$ is a sequence of actions, as follows.

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

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

We define four controller operators by showing their behavior thought semantics rules. We also prove for each of our operators 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).

**Truncation automaton** The operational semantics definition of truncation automata given in [9, 10] is the following:
if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$

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

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 $\tau$. 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 [9, 10]). Its semantics rule states that if $F$ performs the action $a$ and the same action is performed by $E$ (so it is allowed in the current state of the automaton), then $E \triangleright_T F$ performs the action $a$, otherwise it halts.

[2] In [9] internal actions are denoted by $\cdot$. According to the standard notation of process algebras, we use $\tau$ to denote an internal action.

[3] 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.

**Proposition 5.1** *Let* $E^q = \sum_{a \in Act} \begin{cases} a.E^{q'} \text{ iff } \delta(a,q) = q' \\ \mathbf{0} \text{ othw} \end{cases}$
*be the control process 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.*

**Suppression automaton** Referring to [9], it is defined as
$(\mathcal{Q}, q_0, \delta, \omega)$ where $\omega : Act_\tau \times \mathcal{Q} \to \{-, +\}$ indicates whether or not the action in question should be suppressed (-) or emitted (+).
if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$ and $\omega(a, q) = +$

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

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

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

otherwise

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

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 $\tau$. 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 $Act_\tau$ (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 $E$ can perform 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 5.2** *Let* $E^{q,\omega} =$

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

*be the control process 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.*

**Insertion automata** Referring to [9], it is defined as
$(\mathcal{Q}, q_0, \delta, \gamma)$ where $\gamma : Act_\tau \times \mathcal{Q} \to Act_\tau \times \mathcal{Q}$ that specifies the insertion of an action into the sequence of actions of the program. It is necessary to note that in [9, 10] the automaton inserts a finite sequence of actions instead of only one action,

i.e., using the function $\gamma$, it controls if a wrong action is performed. If it happens, the automaton inserts a finite sequence of actions, hence a finite number of intermediate states. Without loss of generality, we consider that it performs only one action. In this way we openly consider all intermediate states. Note that the domain of $\gamma$ is disjoint from the domain of $\delta$ in order to have a deterministic automata.
if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$

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

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

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

otherwise

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

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 $\tau$. We define the controller operators $\triangleright_I$ as follows:

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

where $+a$ is an action not in $Act_\tau$. 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 actio $b$, then the whole system perform $b$. It is possible to note that in the description of insertion automata in [9] the domains of $\gamma$ and $\delta$ are disjoint. In our case, this is guarantee by the premise of the second rule in which we have that $E \overset{a}{\not\longrightarrow} E'$, $E \xrightarrow{+a.b} E'$. In fact for the insertion automata, if a pair $(a, q)$ is not in the domain of $\delta$ and it is in the domain of $\gamma$ it means that if we are in the state $q$ we cannot perform $a$ actions so in order to change state an action different from $a$ must be performed. It is important to note that it is able to insert new actions but it is not able to suppress any action performed by $F$.

**Proposition 5.3** *Let* $E^{q,\gamma} =$

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

*be the control process 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.*

**Edit automata** According to [9], it is defined as $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$ where $\gamma : Act_\tau \times \mathcal{Q} \rightarrow Act_\tau \times \mathcal{Q}$ that specifies the insertion of a finite sequence of actions into the program's actions sequence and $\omega : Act_\tau \times \mathcal{Q} \rightarrow \{-, +\}$ indicates whether or not the

action in question should be suppressed (-) or emitted (+). Also here $\omega$ and $\delta$ have the same domain while the domain of $\gamma$ is disjoint from the domain of $\delta$ in order to have a deterministic automata.
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. We work, without loss of generality, under the additional assumption that $E$ and $F$ never perform the internal action $\tau$. 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 $\triangleright_I$.

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

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

This operator combines the power of the previous two ones.

**Proposition 5.4** *Let* $E^{q,\gamma,\omega} =$

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

*be the control process 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 introduced the control action $-a$ in the semantics of $\triangleright_S$ and $+a$ in the semantics of $\triangleright_I$ in order to find operators that were as similar as possible to suppression and insertion automata, respectively.

# 6 Synthesis of controller program

One of the goals of our work is to find a controller program $Y$ that can secure a given system whatever is $X$. In particular we wonder if there exists an implementation of $Y$ that can be plugged into the system that guarantees the system is secure.

According to the previous section we can enforce safety properties in several ways. As a matter of fact we have described four different controller operators: $Y \triangleright_T X$, $Y \triangleright_S X$, $Y \triangleright_I X$ and $Y \triangleright_E X$. For each of them we want to solve the following problem:

$$\exists Y \; \forall X \quad Y \triangleright_{\mathbf{K}} X \models \phi'$$

where $\mathbf{K}$ is in $\{T, S, I, E\}$ and $\phi'$ is an equational $\mu$-calculus formula as in Formula (5).

For that reason we prove the following proposition.

**Proposition 6.1** *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 $Act_\tau$*[4] *and*

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

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

These operators are applied in order to enforce safety properties. Hence we restrict ourselves to a subclass of equational $\mu$-calculus formulae that is denoted by $Fr_\mu$. This class consists of equational $\mu$-calculus formulae without $\langle \_ \rangle$. It is easy to prove that this set of formulae is closed under the partial model checking function and the following result holds.

**Proposition 6.2** *Let $E$ and $F$ be two finite state processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$.*

At this point in order to satisfy the Formula (5) it is sufficient to find a controller program 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. Hence, for every $\mathbf{K} \in \{T, S, I, E\}$ we calculate $\phi''_{\mathbf{K}} = \phi'_{//_{[f_{\mathbf{K}}]}}$. Thus we obtain:

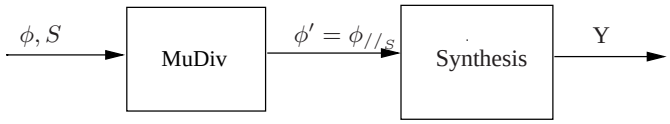
$$\exists Y \quad Y \models \phi''_{\mathbf{K}} \tag{6}$$

This is a satisfiability problem in $\mu$-calculus that can be solved by Theorem 3.1. It is important to note that even if the process $Y$ performs some actions $\tau$ it is possible to obtain from $Y$ another process $Y'$ with only visible actions that is a deterministic model of $\phi$.

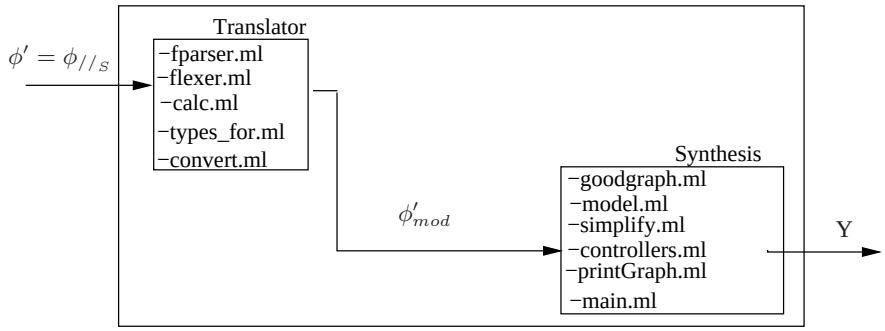
## 6.1 A tool for the Synthesis of Controller Programs

In order to solve the satisfiability problem described by the Formula (6) we have developed a tool that, given a system $S$ and a formula $\phi$, generates a process $Y$. This process is a model for $\phi'$, the formula obtained by the partial evaluation of $\phi$ by $S$ and, moreover, it guarantees that $S \| (Y \triangleright X)$ satisfies $\phi$ whatever $X$ is.

[4] Here the set $Act_\tau$ must be consider enriched by control actions.




a) The architecture of the whole tool




b) A zoom of the Synthesis module

Figure 1: Architecture of the tool.

The tool is made up of two main parts (see Figure 1.a)): the first part implements the partial model checking function; the second one, by implementing the satisfiability procedure developed by Walukiewicz in [47], generates a process $Y$. In particular, it permit to obtain a controller program $Y$ for each controller operators $\triangleright_{\mathbf{K}}$.

In Figure 1 there is a graphical representation of the architecture of the whole tool that we explain in more detail in the following section.

### 6.1.1 Architecture of the tool

The first module of our tool consists in the *MuDiv* module. It implements the partial model checking function. It has been developed in C++ by J.B. Nielsen and H.R. Andersen. The *MuDiv* takes in input a process $S$ described by an $LTS$ and an equational $\mu$-calculus formula $\phi$ and returns an equational $\mu$-calculus formula $\phi' = \phi_{//_S}$.

The second module of our tool is the *Synthesis* module. It is able to build a model for a given modal $\mu$-calculus formula by exploiting the satisfiability procedure developed by Walukiewicz in [47]. It is developed in O'caml 3.09 (see [24]) and it is described better in Figure 1.b) in which we can see that it consists of two submodules: the *Translator* and the *Synthesis*.

**The *Translator*** manages the formula $\phi'$, output of the *MuDiv* module in order to obtain a formula that can be manage 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 formulae instead the partial model checking was developed for equational $\mu$-calculus ones. As a matter of fact, the equational $\mu$-calculus is close for partial model checking. This means that applying the partial model checking function to an equational $\mu$-calculus formula we obtain an equational $\mu$-calculus formula.

The *Translator* consists in four 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*** is an implementation of Walukiewicz satisfiability procedure. Given a modal $\mu$-calculus formula $\phi'_{mod}$ we build a graph by following the set of axioms of the satisfiability procedure of Walukiewicz. For that reason we define the `type graph` as a list of triple $(n, a, n) \in GNode \times Act_\tau \times GNode$ 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 formulae that it satisfies.

The kind of formulae that we consider are formulae that express safety properties, i.e. they are modal $\mu$-calculus formulae without minimum fixpoint and diamond operators.

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 [47] we have to check if the graph that we build is effectively a model or a refutation of $\phi'_{mod}$. We do this by 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 `goodgraph.ml`. It is important to note that the graph that we generate has some transition 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 operators, 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 6.1. In this way we obtain four different processes $Y = Y[f_T]$, because $f_T$ is the identity function on $Act_\tau$, $Y[f_S]$, $Y[f_I]$ and $Y[f_E]$.

Other function in this submodule are the function

`printGraph.ml` that permits to print the graph as a sequence of nodes labeled by a list of formulae, connected by arrows labeled by an action, and the function `main.ml` that calls all the other functions and permits to create the executable file (`.exe`).

# 7 A Cases Study

In order to explain better how the 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 5.1.

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:

$$X =_\nu [\tau]X \wedge [\mathtt{open}]Y$$
$$Y =_\nu [\tau]Y \wedge [\mathtt{close}]X \wedge [\mathtt{open}]F$$

## 7.1 Truncation

We halt the system if the 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 $X = \mathtt{open.open.0}$. Applying $Y \triangleright_T X$ we obtain:

$$Y \triangleright_T X =$$
$$\mathtt{open.close}.Y \triangleright_T \mathtt{open.open.0} \xrightarrow{\mathtt{open}} \mathtt{close}.Y \triangleright_T \mathtt{open.0}$$

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.

## 7.2 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:

$$\begin{array}{l} Y \triangleright_S X = {}^{-}\mathtt{open}.Y + \mathtt{open}.Y' \triangleright_S \mathtt{open.open.}\mathbf{0} \\ \xrightarrow{\mathtt{open}} {}^{-}\mathtt{open}.Y' + \mathtt{close}.Y \triangleright_S \mathtt{open.}\mathbf{0} \xrightarrow{\tau} Y' \triangleright_S \mathbf{0} \end{array}$$

The whole system halts again because, even if a wrong action is suppressed, this controllers cannot introduce right actions.

## 7.3 Insertion

Let $Y$ be a controller program for the controller $Y \triangleright_I X$. We obtain:

$$\begin{array}{l} Y = {}^{+}\mathtt{open.close.open}.Y + \mathtt{open}.Y' \\ Y' = {}^{+}\mathtt{open.close.open}.Y' + \mathtt{close}.Y \end{array}$$

We consider $X$ that tries to open two different files. Hence $X = \mathtt{open.open.}\mathbf{0}$. We obtain:

$$\begin{array}{l} Y \triangleright_I X = \\ {}^{+}\mathtt{open.close.open}.Y + \mathtt{open}.Y' \triangleright_I \mathtt{open.open.}\mathbf{0} \\ \xrightarrow{\mathtt{open}} {}^{+}\mathtt{open.close.open}.Y' + \mathtt{close}.Y \triangleright_I \mathtt{open.}\mathbf{0} \\ \xrightarrow{\mathtt{close}} \mathtt{open}.Y' \triangleright_I \mathtt{open.}\mathbf{0} \xrightarrow{\mathtt{open}} Y' \triangleright_I \mathbf{0} \end{array}$$

We can note the $Y$ permits $X$ to perform the first action $\mathtt{open}$. Then it checks that $X$ is going to perform another $\mathtt{open}$ by the action ${}^{+}\mathtt{open}$. Hence $Y$ insert an action $\mathtt{close}$. After this action it permits $X$ to perform the action $\mathtt{open}$. Since $X$ does not perform any another actions the whole system halts.

## 7.4 Edit

We consider to apply the controller operator $Y \triangleright_E X$. The controller program that we generate is the following:

$$\begin{array}{l} Y = {}^{-}\mathtt{open}.Y + {}^{+}\mathtt{open.close.open}.Y + \ \mathtt{open}.Y' \\ Y' = {}^{-}\mathtt{open}.Y' + {}^{+}\mathtt{open.close.open}.Y' + \mathtt{close}.Y \end{array}$$

We suppose again that $X = \mathtt{open.open.}\mathbf{0}$. We have:

$$\begin{array}{l} Y \triangleright_E X = \\ {}^{-}\mathtt{open}.Y + {}^{+}\mathtt{open.close.open}.Y + \ \mathtt{open}.Y' \triangleright_E \\ \triangleright_E \mathtt{open.open.}\mathbf{0} \xrightarrow{\mathtt{open}} \\ {}^{-}\mathtt{open}.Y' + {}^{+}\mathtt{open.close.open}.Y' + \mathtt{close}.Y \triangleright_E \\ \triangleright_E \mathtt{open.}\mathbf{0} \xrightarrow{\mathtt{close}} \mathtt{open}.Y' \triangleright_E \triangleright_E \mathtt{open.}\mathbf{0} \xrightarrow{\mathtt{open}} Y' \triangleright_E \mathbf{0} \end{array}$$

Also in this case, after the first action $\mathtt{open}$, $Y$ checks if $X$ is going to perform another $\mathtt{open}$ by the action ${}^{+}\mathtt{open}$ and then it inserts the action $\mathtt{close}$ in order to satisfy the property $D$. Then it permit to perform another $\mathtt{open}$ action.

# 8 Further results

## 8.1 Timed setting

In this section we extend to a timed setting the theory that we have previously developed. First of all we show some notions useful to describe a very simple timed setting.

### 8.1.1 *GSOS* and $CCS$ process algebra with time

We follow a simple approach, where time is discrete, actions are durationless and there is one special $tick$ action to represent the elapsing of time (see [37]). These are features of the so called *fictitious clock* approach of, e.g. [14, 21, 45]. A global clock is supposed to be updated whenever all the processes of the system agree on this, by globally synchronizing on action $tick$. Hence, between the two global synchronization on action $tick$ all the processes proceed asynchronously by performing durationless actions. So, the $tick$ action is important in parallel operator whose semantics, in this case, is enriched of this one more rule in addition of rules given in Table 1.

$$\frac{E_1 \stackrel{tick}{\longrightarrow} E_1' \quad E_2 \stackrel{tick}{\longrightarrow} E_2'}{E_1 \| E_2 \stackrel{tick}{\longrightarrow} E_1' \| E_2'}$$

### 8.1.2 Behavioral equivalence

As done in [37] we consider the class of processes that do allow time proceed, the so-called *weakly time alive* processes. These represent *correct* attackers w.r.t. time. (As a matter of fact, it is not realistic that an intruder or a malicious agent can block the flow of time.)

**Definition 8.1** *A process $E$ is* directly weakly time alive *iff $E \stackrel{tick}{\Longrightarrow}$ [5], while it is* weakly time alive *iff for all $E' \in Der(E)$, we have $E'$ is directly weakly time alive.*

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

The behavioral relation considered here is the timed versions of weak bisimulation [33]. This equivalence permits to abstract to some extent from the internal behavior of the systems, represented by the invisible $\tau$ actions.

**Definition 8.2** *Let $(\mathcal{E}, \mathcal{T})$ be an LTS of concurrent processes, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* timed weak simulation*, denoted by $\preceq_t$, over $(\mathcal{E}, \mathcal{T})$ if and only if, whenever $(E, F) \in \mathcal{R}$ we have:*

- *if $E \stackrel{a}{\longrightarrow} E'$ then there exists $F'$ s.t. $F \stackrel{a}{\Longrightarrow} F'$ and $(E', F') \in \mathcal{R}$,*
- *if $E \stackrel{tick}{\longrightarrow} E'$ then there exists $F'$ s.t. $F \stackrel{tick}{\Longrightarrow} F'$ and $(E', F') \in \mathcal{R}$.*

[5] This means that we are no interested to the final state of the transition.

*Moreover, a binary relation $\mathcal{R}$ over $\mathcal{E}$ is said a* timed weak bisimulation *(denoted by $\approx_t$) over the LTS of concurrent processes $(\mathcal{E}, \mathcal{T})$ if both $\mathcal{R}$ and its converse are timed weak simulation.*

### 8.1.3 Partial model checking with time

Introducing the new $tick$ action we have one more case to consider in the definition of partial model checking function. The $tick$ action cannot be consider as the other actions in $Act_\tau$. Hence we extend the partial model checking function to deal with time by adding the following rules

$$\langle tick\rangle A//s = \begin{cases} \langle tick\rangle A//s' & s \xrightarrow{tick} s' \\ \mathbf{F} & otw \end{cases}$$

$$[tick]A//s = \begin{cases} [tick]A//s' & s \xrightarrow{tick} s' \\ \mathbf{T} & otw \end{cases}$$

It is easy to note that the insertion of $tick$ action affects only the partial model checking for parallel operator.

### 8.1.4 Our controller operators in a timed setting

In this section we study how the controller operators that we have defined in Section 5.1 work in a timed setting. We want that $Y \triangleright_{\mathbf{K}} X$, for each $\mathbf{K}$, are processes that do allow time to proceed, so we prove that it is *weakly time alive*. Here we use the following notation: $E$ and $F$ are finite state processes. $E$ is the controller program and $F$ the target. The following proposition holds.

**Proposition 8.1** *If both $E$ and $F$ are weakly time alive, also $E \triangleright_{\mathbf{K}} F$ is weakly time alive.*

Dealing with time does not change or modify the semantic of our controllers. Hence a proposition similar to Proposition 6.1 holds. In particular, looking at the definition of weak timed simulation and at the proof of the Proposition 6.1, given in appendix, the following proposition holds.

**Proposition 8.2** *For every $\mathbf{K} \in \{$truncation, suppression, insertion, edit$\}$ the following relation holds $E \triangleright_{\mathbf{K}} F \preceq_t E[f_{\mathbf{K}}]$ where $f_{\mathbf{K}}$ is a relabeling function definition of which depend on $\mathbf{K}$.*

We can then recast results of the previous section in a timed setting.

## 8.2 Parameterized Systems

A parameterized system describes an infinite family of (typically finite-state) systems; instances of the family can be obtained by fixing parameters. 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 8.1** *Consider the system with one consumer process $C$ and $n$ producer processes $P$. Each process $P$ is defined $P \stackrel{def}{=} a.P$ where $a \in Act$, and the process $C$ is $\bar{a}.C$. 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 (2) we have

$$\forall n \quad \forall X \quad P_n \| X \models \phi \tag{7}$$

It is possible to note that in the previous equation there are two universal quantifications; the first one on the number of instances of the process $P$ and the second one on the possible unknown agents.

In order to eliminate the universal quantification on the number of processes, first of all, we define the concept of *invariant formula w.r.t. partial model checking for parallel operator* as follows.

**Definition 8.3** *A formula $\phi$ is said an* invariant w.r.t. partial model checking *for the system $P \| X$ iff $\phi \Leftrightarrow \phi_{//_P}$.*

It is possible to prove the following result.

**Proposition 8.3** *Given the system $\forall i \; P_i \| X$. If $\phi$ is an invariant formula for this system then*

$$\forall X \;\; (\forall n \;\; P_n \| X \models \phi \quad \textit{iff} \quad X \models \phi)$$

In order to apply the theory developed in Section 4, we show a method to find the invariant formula. According to [8], let $\psi_i$ be defined as follows

$$\psi_i = \begin{cases} \phi'_1 & \text{if } i = 1 \\ \psi_{i-1} \wedge \phi'_i & \text{if } i > 1 \end{cases}$$

By definition of $\psi_i$ and by Lemma 3.3, $\forall j$ s.t. $1 \leq j \leq i \; (X \models \phi'_j) \Leftrightarrow X \models \psi_i$. Hence $X \models \psi_i$ means that $\forall j$ s.t. $1 \leq j \leq i \; P_j \| X \models \phi'$. 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 formulae, that is also said *limit of the sequence*, then the following identity holds.

$$\forall X \; (X \models \psi_\omega \Leftrightarrow \forall n \geq 1 \; P_n \| X \models \phi') \tag{8}$$

Now we can apply the reasoning made in Section 4. Hence we are able to define a controller operator that forces each process to behave correctly and synthesize a controller program.

In some cases it could not be possible to find the limit of the chain. However there are some technique that can be useful in order to find an approximation of this limit (see [8, 15]).

## 8.3 Composition of safety properties

Our logical approach is able to struggle successfully with composition problems by using the operator $\triangleright_T$. We suppose to have to force a systems to satisfy a security policy that can be write as the conjunction of several safety properties as follows:

$$\forall X \quad S \| X \models \phi = \phi_1 \wedge \ldots \wedge \phi_n \tag{9}$$

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 $Y$ for a given controller operator that force $\phi$ to be satisfied. So we want to find $Y$ s.t.:

$$\forall X \quad S \| Y \triangleright_T X \models \phi_1 \wedge \ldots \wedge \phi_n \tag{10}$$

According to Theorem 3.1, the cost of the satisfiability procedure is exponential in the size of the formula. What we prove here is a method to find a controller program $Y$ for $\phi$ starting from controller operators for safety formula simpler than $\phi$. To do this we split $\phi$ in a finite number $n$ of sub-formulae, whenever it is possible, $\phi_1, \ldots, \phi_n$, s.t. $\phi = \bigwedge_{i=1}^{n} \phi_i$. Then, by exploiting he Theorem 3.1, 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, synthesize $Y$ is exponential in the size of $\phi$. Let we consider that all the $\phi_i$ have the same size $m$ and let the cost of the composition be constant. Then the cost of our method is $n\mathcal{O}(2^m)$ instead of $\mathcal{O}(2^{m \times n})$.

In order to describe our method, first of all, we rewrite Formula (9), 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 8.1** *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 s.t. $\forall i$ s.t. $1 \leq i \leq n$ $Y_i \models \phi_i$. 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, if we have several controller programs for several safety properties, applying them one after the other we can enforce a safety property that is the conjunction of the previous ones. However, in this way, we apply the procedure for enforcing $n$ times. Instead we want apply it only one time to force that conjunction of formulae. For that reason we prove the following proposition.

**Proposition 8.4** *Let we consider the controller operator $\triangleright_T$. It is possible to find $Y_1, \ldots, Y_n$ controller programs s.t. 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 10, 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 8.4. In this way we find $Y$ that force $\phi' = \phi'_1 \wedge \ldots \wedge \phi'_n$. According to Lemma 3.3 we have:

$$\begin{array}{c} \forall X \; Y \triangleright_T X \models \phi' \\ \Updownarrow \\ \forall X \; S \| Y \triangleright_T X \models \phi \end{array}$$

Hence we obtain a controller program $Y$ for $\phi$.

It is important to note that the Proposition 8.4 holds only for the operator $\triangleright_T$ because, as it is possible to see from the proof in Appendix A, it is necessary that both processes, the controller and the target, agree on the action are going to perform. Looking to the operational semantics of controller operators, it is easy to see that the operator $\triangleright_T$ is the only one that satisfies this requirement.

# 9 Conclusion

We illustrated some results towards a uniform theory for enforcing security properties. With this work, we extended a framework based on process calculi and logical techniques, that have been shown to be very suitable to model and verify several security properties, to tackle also synthesis problems of secure systems. In particular we have shown how security properties can be conveniently *specified* and *verified* in a uniform way by using a few concepts of concurrency and temporal logic theory, as, for instance, partial model checking. Using the same framework we also deal with the *synthesis* of secure systems.

Moreover we have described a tool for the synthesis of a controller program based on Walukiewicz's satisfiability procedure as well as on the partial model checking technique. In particular, starting from a system $S$ and a formula $\phi$ that describes a security property, the tool generates a process that, by monitoring a possible un-trusted component, guarantees that a system $S \| X$ satisfies $\phi$ whatever $X$ is.

We also deal with the synthesis of secure systems in a timed setting and for parameterized systems. We present also a method to enforce composition of policies.

# A Technical proofs

**Lemma 3.2** *Let $E$ be a finite-state process and let $\phi_{E,\preceq}$ be its characteristic formula w.r.t. weak simulation.*

$$F \preceq E \Leftrightarrow F \models \phi_{E,\preceq}$$

*Proof*: In order to prove the following proposition we give the following chain:

$$F \preceq E \Leftrightarrow \forall \alpha \;\; F \stackrel{\alpha}{\rightarrow} F' \;\; \exists E' \;\; E \stackrel{\alpha}{\Rightarrow} E' \wedge F' \preceq E' \Leftrightarrow$$
$$\forall \alpha \;\; F \stackrel{\alpha}{\rightarrow} F' \;\; F' \models \bigvee X_{E'} \Leftrightarrow \forall \alpha \;\; F \models [\alpha](\bigvee X_{E'}) \Leftrightarrow$$
$$F \models \bigwedge([\alpha](\bigvee X_{E'}))$$

■

In order to guarantee homogeneity of notation, we assume to work with an $LTS$, since that both automata and sequential process are $LTS$ (see [33]). We should give a proof that a bisimulation exists between automata and controller operator so they have the same behavior.

Before starting to prove Propositions 5.1, 5.2, 5.3, 5.4, we note that in our controller operators the halt condition is not roundly given because this occurs when there are not rule that could be applied, i.e., when premises of all rules are not verify. As we have already note, also in security automata described in Section 5.1, the action $\tau$ in stop rule of each automata 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.

**Proposition 5.1** Let $E^q = \sum_{a \in Act} \begin{cases} a.E^{q'} \text{ iff } \delta(a,q) = q' \\ \mathbf{0} \text{ othw} \end{cases}$ be the control process 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:* We can define the relation of strong bisimulation $\mathcal{R}_T$ in the following way:

$$\mathcal{R}_T = \{((\sigma, q), E^q \triangleright_T F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \stackrel{\sigma}{\mapsto}\}$$

Assume that $(\sigma, q) \stackrel{a}{\longrightarrow}_T (\sigma', q')$. For the semantic rule of $\triangleright_T$, if $E^q \stackrel{a}{\longrightarrow} E^{q'}$ and $F \stackrel{a}{\longrightarrow} F'$ perform the action $a$ also $E^q \triangleright_T F \stackrel{a}{\longrightarrow} E^{q'} \triangleright_T F'$ and $F' \stackrel{\sigma'}{\mapsto}$. Now assume that $E^q \triangleright_T F \stackrel{a}{\longrightarrow} E^{q'} \triangleright_T F'$ and $F' \stackrel{\sigma'}{\mapsto}$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \stackrel{a}{\longrightarrow}_T (\sigma, q)'$ and $(E^{q'} \triangleright_T F', (\sigma, q)') \in \mathcal{R}_T$. For the rule T-Step, $(\sigma, q) \stackrel{a}{\longrightarrow}_T (\sigma', q')$. So the couple that we are looking for is $(\sigma', q')$. ■

**Proposition 5.2** Let $E^{q,\omega} =$

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

be the control process 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.

*Proof:* The scheme of the proof and the notation are the same of the previous one. Let

$$\mathcal{R}_S = \{((\sigma, q), E^{q,\omega} \triangleright_S F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \overset{\sigma}{\mapsto}\}$$

be the strong bisimulation relation. We have two cases: the first one is similar of proposition 5.1 in fact, let $((\sigma, q), E^{q,\omega} \triangleright_S F)$ be in $\mathcal{R}_S$ and $(\sigma, q) \xrightarrow{a}_S (\sigma', q')$. We should prove that exists a $(E^{q,\omega} \triangleright_S F)'$ s.t. $E^{q,\omega} \triangleright_S F \xrightarrow{a} (E^{q,\omega} \triangleright_S F)'$ and $((\sigma', q'), (E^{q,\omega} \triangleright_S F)') \in \mathcal{R}_S$. By the first rule of $\triangleright_S$ and by definition of $E^{q,\omega}$, using a similar reason of the proof of proposition 5.1, we trivially have the thesis. On the other hand, let $(E^{q,\omega} \triangleright_S F, (\sigma, q))$ be in $\mathcal{R}_S$ and $E^{q,\omega} \triangleright_S F \xrightarrow{a} E^{q',\omega} \triangleright_S F'$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \xrightarrow{a}_S (\sigma, q)'$ and $(E^{q',\omega} \triangleright_S F', (\sigma, q)') \in \mathcal{R}_S$. For the rule S-StepA we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the previous one.

Now, let $((\sigma, q), E^{q,\omega} \triangleright_S F)$ be in $\mathcal{R}_S$ and $(\sigma, q) \xrightarrow{\tau}_S (\sigma', q')$. We should prove that exists a $(E^{q,\omega} \triangleright_S F)'$ s.t. $E^{q,\omega} \triangleright_S F \xrightarrow{\tau} (E^{q,\omega} \triangleright_S F)'$ and $((\sigma', q'), (E^{q,\omega} \triangleright_S F)') \in \mathcal{R}_S$. We have, by the second rule of $\triangleright_S$ and by the definition of $E^{q,\omega}$, that if $E^{q,\omega} \xrightarrow{-a} E^{q',\omega}$ and $F \xrightarrow{a} F'$ then $E^{q,\omega} \triangleright_S F \xrightarrow{\tau} E^{q',\omega} \triangleright_S F'$. We have also $F' \overset{\sigma'}{\mapsto}$.So $((\sigma', q'), E^{q',\omega} \triangleright_S F') \in \mathcal{R}_S$ trivially.

Now assume that $(E^{q,\omega} \triangleright_S F, (\sigma, q))$ be in $\mathcal{R}_S$ and $E^{q,\omega} \triangleright_S F \xrightarrow{\tau} E^{q',\omega} \triangleright_S F'$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \xrightarrow{\tau}_S (\sigma, q)'$ and $(E^{q',\omega} \triangleright_S F', (\sigma, q)') \in \mathcal{R}_S$ 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. ■

**Proposition 5.3**Let $E^{q,\gamma} =$

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

be the control process 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.

*Proof:* The scheme of the proof and the notation are the same of the previous one.Let $\mathcal{R}_I$ be the strong bisimulation relation defined as follows:

$$\mathcal{R}_I = \{((\sigma, q), E^{q,\gamma} \triangleright_I F) : (\sigma, q) \in \overrightarrow{Act} \times Q, F \overset{\sigma}{\mapsto}\}$$

We have two cases: the first one is similar of proposition 5.1 in fact, let $((\sigma, q), E^{q,\gamma} \triangleright_I F)$ be in $\mathcal{R}_I$ and $(\sigma, q) \xrightarrow{a}_I (\sigma', q')$. We should prove that exists a $(E^{q,\gamma} \triangleright_I F)'$ s.t. $E^{q,\gamma} \triangleright_I F \xrightarrow{a} (E^{q,\gamma} \triangleright_I F)'$ and $((\sigma', q'), (E^{q,\gamma} \triangleright_I F)') \in \mathcal{R}_I$. By the first rule of $\triangleright_I$ and by definition of $E^{q,\gamma}$,using a similar reasoning of the proof of proposition 5.1, we trivially have the thesis. On the other hand, let $(E^{q,\gamma} \triangleright_I F, (\sigma, q))$ be in $\mathcal{R}_I$ and $E^{q,\gamma} \triangleright_I F \xrightarrow{a} E^{q',\gamma} \triangleright_I F'$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \xrightarrow{a}_I (\sigma, q)'$ and $(E^{q',\gamma} \triangleright_I F', (\sigma, q)') \in \mathcal{R}_I$. 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.

Now let $((\sigma, q), E^{q,\gamma} \triangleright_I F)$ be in $\mathcal{R}_I$ and $(\sigma, q) \xrightarrow{b}_I (\sigma, q')$. We should prove that exists a $(E^{q,\gamma} \triangleright_I F)'$ s.t. $E^{q,\gamma} \triangleright_I F \xrightarrow{b} (E^{q,\gamma} \triangleright_I F)'$ and $((\sigma, q'), (E^{q,\gamma} \triangleright_I F)') \in \mathcal{R}_I$.

We have, by second rule of $\triangleright_I$ and by to the definition of $E^{q,\gamma}$, that if $E^{q,\gamma} \stackrel{a}{\not\longrightarrow} E^{q',\gamma}$, $E^{q,\gamma} \stackrel{+a.b}{\longrightarrow} E^{q',\gamma}$ and $F \stackrel{a}{\longrightarrow} F'$ then $E^{q,\gamma} \triangleright_I F \stackrel{b}{\longrightarrow} E^{q',\gamma} \triangleright_I F$. So $(E^{q,\gamma} \triangleright_I F)'$ is $E^{q',\gamma} \triangleright_I F$ and $((\sigma, q'), E^{q',\gamma} \triangleright_I F) \in \mathcal{R}_I$ trivially.

Now, let $(E^{q,\gamma} \triangleright_I F, (\sigma, q))$ be in $\mathcal{R}_I$ and $E^{q,\gamma} \triangleright_I F \stackrel{b}{\longrightarrow} E^{q',\gamma} \triangleright_I F$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \stackrel{b}{\longrightarrow} (\sigma, q)'$ and $(E^{q',\gamma} \triangleright_I F, (\sigma, q)') \in \mathcal{R}_I$. 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 5.4**Let $E^{q,\gamma,\omega} =$

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

be the control process 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.

*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 $\triangleright_E$ as follows:

$$\mathcal{R}_E = \{((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F) : (\sigma, q) \in \overrightarrow{Act} \times Q, E^{q,\gamma,\omega} \triangleright_E F \in \mathcal{P},$$

$$F \stackrel{\sigma}{\mapsto}\}$$

We have three cases ad their proof following the reasoning made in the proof of lemma 5.2 and lemma 5.3. In fact:

- – Let $((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F)$ be in $\mathcal{R}_E$ and $(\sigma, q) \stackrel{a}{\longrightarrow}_E (\sigma', q')$. We should prove that exists a $(E^{q,\gamma,\omega} \triangleright_E F)'$ s.t. $E^{q,\gamma,\omega} \triangleright_E F \stackrel{a}{\longrightarrow}_E (E^{q,\gamma,\omega} \triangleright_E F)'$ and $((\sigma', q'), (E^{q,\gamma,\omega} \triangleright_E F)') \in \mathcal{R}_E$. We have, by the first rule of $\triangleright_E$ and by definition of $E^{q,\gamma,\omega}$, that if $E^{q,\gamma,\omega} \stackrel{a}{\longrightarrow}_E E^{q',\gamma,\omega}$ and $F \stackrel{a}{\longrightarrow} F'$ then $E^{q,\gamma,\omega} \triangleright_E F \stackrel{a}{\longrightarrow} E^{q',\gamma,\omega} \triangleright_E F'$. Now $F' \stackrel{\sigma}{\mapsto}'$. So $(E^{q,\gamma,\omega} \triangleright_E F)'$ is $E^{q',\gamma,\omega} \triangleright_E F'$ and $((\sigma', q'), E^{q',\gamma,\omega} \triangleright_E F') \in \mathcal{R}_E$ trivially.
  - – Let $(E^{q,\gamma,\omega} \triangleright_E F, (\sigma, q))$ be in $\mathcal{R}_E$ and $E^{q,\gamma,\omega} \triangleright_E F \stackrel{a}{\longrightarrow} E^{q',\gamma,\omega} \triangleright_E F'$. We should prove that exists a $(\sigma, q)'$ s.t. $(\sigma, q) \stackrel{a}{\longrightarrow} (\sigma, q)'$ and $(E^{q',\gamma,\omega} \triangleright_E F', (\sigma, q)') \in \mathcal{R}_E$. For the rule E-StepA we have that $(\sigma', q')$ is the solution we are looking for. The reasoning is similar to the previous one.
- – Let $((\sigma, q), E^{q,\gamma,\omega} \triangleright_E F)$ be in $\mathcal{R}_E$ and $(\sigma, q) \stackrel{\tau}{\longrightarrow}_E (\sigma', q')$. We should prove that exists a $(E^{q,\gamma,\omega} \triangleright_E F)'$ s.t. $E^{q,\gamma,\omega} \triangleright_E F \stackrel{\tau}{\longrightarrow} (E^{q,\gamma,\omega} \triangleright_E F)'$ and $((\sigma', q'), (E^{q,\gamma,\omega} \triangleright_E F)') \in \mathcal{R}_E$. We have, by second rule of $\triangleright_E$ and by the definition of $E^{q,\gamma,\omega}$, that if $E^{q,\gamma,\omega} \stackrel{-a}{\longrightarrow} E^{q',\gamma,\omega}$ and $F \stackrel{a}{\longrightarrow} F'$ then $E^{q,\gamma,\omega} \triangleright_E F \stackrel{\tau}{\longrightarrow} E^{q',\gamma,\omega} \triangleright_E F'$. Now $F' \stackrel{\sigma}{\mapsto}'$. So $(E^{q,\gamma,\omega} \triangleright_E F)'$ is $E^{q',\gamma,\omega} \triangleright_E F'$ and $((\sigma', q'), E^{q',\gamma,\omega} \triangleright_E F') \in \mathcal{R}_E$ trivially.

– 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 should prove that exists a $(\sigma, q)'$ s.t.
$(\sigma, q) \xrightarrow{\tau}_e (\sigma, q)'$ and $(E^{q,\gamma,\omega} \triangleright_E F', (\sigma, q)') \in \mathcal{R}_E$ 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 should prove that exists a $(E^{q,\gamma,\omega} \triangleright_E F)'$ s.t. $E^{q,\gamma,\omega} \triangleright_E F \xrightarrow{b} (E^{q,\gamma,\omega} \triangleright_E F)'$ and $((\sigma, q'), (E^{q,\gamma,\omega} \triangleright_E F)') \in \mathcal{R}_E$. We have, by third rule of $\triangleright_E$ and by the definition of $E^{q,\gamma,\omega}$ that if $E^{q,\gamma,\omega} \not\xrightarrow{a} E^{q',\gamma,\omega}$, $E^{q,\gamma,\omega} \xrightarrow{+a.b} E^{q',\gamma,\omega}$ and $F \xrightarrow{a} F'$ 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 $((\sigma, q'), E^{q',\gamma,\omega} \triangleright_E F) \in \mathcal{R}_E$ trivially.

  – 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 should prove that exists a $(\sigma, q)'$ s.t.
  $(\sigma, q) \xrightarrow{b} (\sigma, q)'$ and $(E^{q',\gamma} \triangleright_E F, (\sigma, q)') \in \mathcal{R}_E$. 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.

■

**Proposition 6.1** For every $\mathcal{K} \in \{\text{truncation, suppression, insertion, edit}\}$ the following relation holds

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

where $f_{\mathcal{K}}$ is a relabeling function definition of which depend on $\mathcal{K}$.

In order to prove this proposition we prove the following four lemmas. The proof of the proposition comes trivially from the union of the proof of the lemmas.

**Lemma A.1** *The following relation holds*

$$Y \triangleright_T X \preceq Y[f_T] \tag{11}$$

*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 $E \stackrel{\alpha}{\Rightarrow} E'$ and, obviously, $(E' \triangleright_T F', E') \in \mathcal{S}_T$. ■

**Lemma A.2** *The following relation holds*

$$Y \triangleright_S X \preceq Y[f_S] \tag{12}$$

*where*

$$f_S(a) = \begin{cases} a & \text{if } a \in Act \\ \tau & \text{if } a = -a \end{cases}$$

*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. If $E \triangleright_S F \xrightarrow{\tau} E' \triangleright_S F'$ means that $E \xrightarrow{-a} E'$ and $F$ perform an action $a$ that $E$ should not perform. Applying the relabeling function $f_S$ to $E$ we obtain $E_1 = E[f_S]$ s.t. $E_1 \stackrel{\tau}{\Longrightarrow} E_1'$. where $E_1'$ is $E'[f_S]$. Hence $(E' \triangleright_S F', E_1') \in \mathcal{S}_S$. ■

**Lemma A.3** *The following relation holds*

$$Y \triangleright_I X \preceq Y[f_I] \tag{13}$$

*where*

$$f_I(a) = \begin{cases} a & \text{if } a \in Act \\ \tau & \text{if } a = +a \end{cases}$$

*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. If $E \triangleright_I F \xrightarrow{b} E' \triangleright_I F$ means that $E \xrightarrow{+a.b} E'$ and $F$ perform 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 $E_1 = E[f_I]$ s.t. $E_1 \stackrel{b}{\Longrightarrow} E_1'$. where $E_1'$ is $E'[f_I]$. Hence $(E' \triangleright_I F', E'1) \in \mathcal{S}_I$. ■

**Lemma A.4** *The following relation holds*

$$Y \triangleright_E X \preceq Y[f_E] \tag{14}$$

*where*

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

*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 is 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 case is the following:

- $E \triangleright_E F \xrightarrow{\tau} E' \triangleright_E F'$ we want to find a $E'[f_E]$ s.t. $E[f_E] \xrightarrow{\tau} E[f_E]'$. Referring to the second rule of the edit automata we see that $E \triangleright_E F \xrightarrow{\tau} E' \triangleright_E F'$ when $E \xrightarrow{-a} E'$. Through the relabeling function $f_E$ we have $E[f_E] \xrightarrow{\tau} E'[f_E]$ and $(E' \triangleright_E F', E'[f_E]) \in \mathcal{S}_E$.

- $E \rhd_E F \xrightarrow{b} E' \rhd_E F$ we want to find a $E'[f_E]$ s.t. $E[f_E] \stackrel{b}{\Longrightarrow} E[f_E]'$. Referring to the last rule of edit automata we see that $E \rhd_E F \xrightarrow{b} E' \rhd_E F$ when $E \xrightarrow{+a.b} E'$. Through the relabeling function $f_E$ we have $E[f_E] \stackrel{b}{\Longrightarrow} E'[f_E]$ and $(E' \rhd_E F, E'[f_E]) \in \mathcal{S}_E$

■

**Proposition 6.2** *Let $E$ and $F$ be two finite state processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$.*

*Proof*: A translation from equational $\mu$-calculus to modal $\mu$-calculus is possible. So first of all we consider the modal formula associated with the given formula $\phi$ then the proof may be divided in two part. Former we prove the proposition holds for the formulae of modal $\mu$-calculus without recursion operator, latter we extended the results also to $\mu X.\phi$ and $\nu X.\phi$.

The first part is very similar to the proof proposed by Stirling in [43] that is made by induction on the structure of the formula $\phi$. The base case is clear. For the inductive step first suppose $\phi = \phi_1 \wedge \phi_2$ and that the result holds for the components $\phi_1$ and $\phi_2$. By the definition of satisfaction relation $E \models \phi$ iff $E \models \phi_1$ and $E \models \phi_2$. By inductive hypothesis $F \models \phi_1$ and $F \models \phi_2$ then $F \models \phi$. A similar argument justifies the case $\phi = \phi_1 \vee \phi_2$. Next suppose $\phi = [a]\phi_1$ and $E \models \phi$. Therefore for any $E'$ s.t. $E \stackrel{a}{\Rightarrow} E'$ it follows that $E' \models \phi_1$. Let $F \xrightarrow{a} F'$ we know that for some $E'$ there is the transition $E \stackrel{a}{\Rightarrow} E'$ and $F' \preceq E'$, so by inductive hypothesis $F' \models \phi_1$ and so $F \models \phi$. Now we have to prove that if $\phi = \mu X.\phi_1$ or $\phi = \nu X.\phi_1$ the proposition holds. Referring to the definition of minimum and maximum fixed point we can consider these as inductive limit (the union) of formulae like $\mu X^\alpha.\phi_1$, where $\mu X^0.\phi_1 = \mathbf{F}$ and $\mu X^{\alpha+1}.\phi_1 = \phi_1[\mu X^\alpha.\phi_1/X]$, and $\nu X^\alpha.\phi_1$ where $\nu X^0.\phi_1 = \mathbf{T}$ and $\nu X^{\alpha+1}.\phi_1 = \phi_1[\nu X^\alpha.\phi_1/X]$. In this way $E \models \mu X.\phi_1$ iff $E \models \mu X^\alpha.\phi_1$ for some $\alpha$ iff $E \models \bigvee_\alpha(\mu X^\alpha.\phi_1)$ and $E \models \nu X.\phi_1$ iff $E \models \nu X^\alpha.\phi_1$ for all $\alpha$ iff $E \models \bigwedge_\alpha(\nu X^\alpha.\phi_1)$. In the former case we have a sequence of disjunction and in the latter we have a sequence of conjunction. We can apply again the argument of the first part of the proof. ■

**Proposition 8.1**: Let $E$ and $F$ be two finite-state processes. If both $E$ and $F$ are weakly time alive, also $E \rhd_\mathcal{K} F$ is weakly time alive.

In order to prove this proposition we prove four lemmas, one for each of the four operators.

**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 $(E \rhd_T F)' \in Der(E \rhd_T F)$ $(E \rhd_T F)' \stackrel{tick}{\Longrightarrow}$. $E$ and $F$ are time alive so

- for all $E' \in Der(E)$ $E' \stackrel{tick}{\Longrightarrow}$
- for all $F' \in Der(F)$ $F' \stackrel{tick}{\Longrightarrow}$

So $\exists E', F'$ such that $(E \rhd_T F)' = E' \rhd_T F'$ and, referring to the semantic rule of $\rhd_T$ $E' \rhd_T F' \stackrel{tick}{\Longrightarrow}$ ■

**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 $(E \rhd_I F)' \in Der(E \rhd_I F)$ $(E \rhd_I F)' \stackrel{tick}{\Longrightarrow}$. $E$ and $F$ are time alive so

- for all $E' \in Der(E)$ $E' \stackrel{tick}{\Longrightarrow}$
- for all $F' \in Der(F)$ $F' \stackrel{tick}{\Longrightarrow}$

We have two cases: if the first semantic rule is applied $(E \rhd_I F)' = E' \rhd_I F'$ and the prove is the same of the previous lemma. If the second rule is applied we have $(E \rhd_I F)' = E' \rhd_I F$. Noting that $F \in Der(F)$ we can follow the same reasoning do before. ■.

**Lemma A.8** *If both $E$ and $F$ are weakly time alive, also $E \rhd_E F$ is weakly time alive.*

*Proof*: The cases the could be happened here are the same of the lemma A.6 and lemma A.7. So we omit it. ■.

**Proposition 8.4**: Let we consider the controller operator $\rhd_T$.It is possible to find $Y_1, \ldots Y_n$ controller programs s.t. if $Y_1 \rhd_T X \models \phi_1, \ldots, Y_n \rhd_T X \models \phi_n$ then $(Y_1 \rhd_T \ldots \rhd_T Y_n) \rhd_T X \models \phi_1 \wedge \ldots \wedge \phi_n$.

In order to prove the previous proposition we prove some lemmas.

**Lemma A.9** *The following relation holds*

$$Y \rhd_T X \preceq X \tag{15}$$

*Proof:* We prove that the following relation is a weak simulation.

$$\mathcal{S} = \{(E \rhd_T F, F) | E, F \in \mathcal{E}\}$$

Assume that $E \rhd_T F \xrightarrow{a} E' \rhd_T F'$ with the additional hypothesis that $F \xrightarrow{a} F'$ then, by the rule of $\rhd_T$ we have that $E \stackrel{\alpha}{\Rightarrow} E'$ and, obviously, $(E' \rhd_T F', F') \in \mathcal{S}$. ■

**Lemma 8.1**: 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 s.t. $\forall i$ s.t. $1 \leq i \leq n$ $Y_i \models \phi_i$. 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 formulae in the conjunction $n$:

$n = 1$**:** In this case $\phi = \phi_1$. Hence, by exploiting the satisfiability procedure we obtain $Y = Y_1$ that is the controller program s.t. $Y \rhd_T X \models \phi$.

$n \Rightarrow n + 1$**:** Let $\phi$ be a formula s.t. $\phi = \phi_1 \wedge \ldots \wedge \phi_{n+1}$ and $Y_{n+1}$ be a controller program s.t. $\forall X$ $Y_{n+1} \rhd_T X \models \phi_{n+1}$. For inductive hypothesis we know that $\forall X \quad Y_n \rhd_T (Y_{n-1} \rhd_T (\ldots \rhd_T (Y_2 \rhd_T (Y_1 \rhd_T X)))) \models \phi_1 \wedge \ldots \wedge \phi_n$. We have to prove that $\forall X \quad Y_{n+1} \rhd_T (Y_n \rhd_T (Y_{n-1} \rhd_T (\ldots \rhd_T (Y_2 \rhd_T (Y_1 \rhd_T X))))) \models \phi$.

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 $\forall X \quad Y_{n+1} \triangleright_T X \models \phi_{n+1}$, so $Y_{n+1} \triangleright_T Y^n \models \phi_{n+1}$. For Lemma 6.2 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 $Y_{n+1} \triangleright_T Y^n \models \phi$.

■

**Lemma A.10** *Let $\phi, Y_1, \ldots, Y_n$ be as in Lemma 8.1. 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 8.1 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 formulae (see [43]). 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 8.4*: It follows directly from proofs of Lemma 8.1 and Lemma A.10. ■

# References


[1] Aceto, L., Bloom, B., Vaandrager, F.: Turning SOS rules into equations. Information and Computation **111**(1), 1–52 (1994)

[2] Andersen, H.R.: Partial model checking (extended abstract). In: Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science, pp. 398–407. IEEE Computer Society Press (1995)

[3] Arnold, A., Vincent, A., Walukiewicz, I.: Games for synthesis of controllers with partial observation. Theoretical Computer Science **303**(1), 7–34 (2003)

[4] Badouel, E., Caillaud, B., Darondeau, P.: Distributing finite automata through petri net synthesis. Journal on Formal Aspects of Computing **13**, 447–470 (2002)

[5] Bartoletti, M., Degano, P., Ferrari, G.: Policy framings for access control. In: Proceedings of the 2005 workshop on Issues in the theory of security, pp. 5 – 11. Long Beach, California (2005)

[6] Bartoletti, M., Degano, P., Ferrari, G.L.: Enforcing secure service composition. In: CSFW, pp. 211–223. IEEE Computer Society (2005)

[7] Basin, D., Olderog, E.R., Sevinç, P.E.: Specifying and analyzing security automata using csp-oz. In: AsiaCCS 2007. ACM, ACM (2007). URL http://www.zisc.ethz.ch/research/publications

[8] Basu, S., Ramakrishnan, C.R.: Compositional analysis for verification of parameterized systems. In: Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 2619, pp. 315–330. Springer, Warsaw, Poland (2003)

[9] Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: I. Cervesato (ed.) Foundations of Computer Security: proceedings of the FLoC'02 workshop on Foundations of Computer Security, pp. 95–104. DIKU Technical Report, Copenhagen, Denmark (2002)

[10] Bauer, L., Ligatti, J., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security **4**(1–2), 2–16 (2005)

[11] Bhat, G., Cleaveland, R.: Efficient model checking via the equational $\mu$-calculus. In: Proceedings, $11^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science, pp. 304–312. IEEE Computer Society Press, New Brunswick, New Jersey (1996)

[12] Bloom, B.: Structured operational semantics as a specification language. In: Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), pp. 107–117. ACM Press, San Francisco, California (1995)

[13] Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. J.ACM **42**(1) (1995)

[14] Corradini, F., D'Ortenzio, D., Inverardi, P.: On the relationships among four timed process algebras. Fundam. Inform. **38**(4), 377–395 (1999)

[15] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for staticanalysis of programs by construction or approximation offixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)

[16] Elmqvist, J., Nadjm-Tehrani, S., Minea, M.: Safety interfaces for component-based systems. In: R. Winther, B.A. Gran, G. Dahll (eds.) SAFECOMP, *Lecture Notes in Computer Science*, vol. 3688, pp. 246–260. Springer (2005)

[17] Emerson, E.A.: Temporal and modal logic pp. 995–1072 (1990)

[18] Focardi, R., Gorrieri, R.: A classification of security properties for process algebras. Journal of Computer Security **3**(1), 5–33 (1994/1995)

[19] Havelund, K., Rosu, G.: 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, pp. 342–356. Springer-Verlag, London, UK (2002)

[20] Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985). DOI http://doi.acm.org/10.1145/2455.2460

[21] Hennessy, M., Regan, T.: 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, pp. 33–48. North-Holland (1991)

[22] Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open systems in reactive environments: Control and synthesis. Lecture Notes in Computer Science **1877**, 92+ (2000). URL citeseer.ist.psu.edu/kupferman00open.html

[23] Kupferman, O., Vardi, M.: $\mu$-calculus synthesis. In: Proc. 25th International Symposium on Mathematical Foundations of Computer Science, *Lecture Notes in Computer Science*, vol. 1893, pp. 497–507. Springer-Verlag (2000)

[24] Leroy, X., Damien Doligez Jacques Garrigue, D.R., Vouillon, J.: The objective caml systemrelease 3.09 (2004). URL http://caml.inria.fr/pub/docs/manual-ocaml/index.html

[25] Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: 10th European Symposium on Research in Computer Security (ESORICS) (2005). URL http://www.cs.princeton.edu/ jligatti/papers/nonsafety.pdf

[26] Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (extended abstract). URL citeseer.ist.psu.edu/302111.html

[27] Martinelli, F.: Formal Methods for the Analysis of Open Systems with Applications to Security Properties. Ph.D. thesis, University of Siena (1998)

[28] Martinelli, F.: Towards automatic synthesis of systems without informations leaks. In: Proceedings of Workshop in Issues in Theory of Security (WITS) (2000)

[29] Martinelli, F.: Analysis of security protocols as open systems. Theoretical Computer Science **290**(1), 1057–1106 (2003)

[30] Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. In: Proceedings of ENTCS STM06 (2006)

[31] Matteucci, I.: A tool for the synthesis of programmable controllers. In: Proceedings of FAST 2006

[32] Milner, R.: Operational and algebraic semantics of concurrent processes. In: J. van Leewen (ed.) Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics, chap. 19, pp. 1201–1242. The MIT Press, New York, N.Y. (1990)

[33] Milner, R.: Communicating and mobile systems: the $\pi$-calculus. Cambridge University Press (1999)

[34] Müller-Olm, M.: Derivation of characteristic formulae. In: MFCS'98 Workshop on Concurrency, *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 18. Elsevier Science B.V. (1998)

[35] Pinchinat, S., Riedweg, S.: A decidable class of problems for control under partial observation. pp. 454–460 (2005)

[36] Raclet, J., Pinchinat, S.: The control of non-deterministic systems: a logical approach. In: Proc. 16th IFAC Word Congress. Prague, Czech Republic (2005)

[37] R.Focardi, R.Gorrieri, F.Martinelli: Real-time Information Flow Analysis. IEEE JSAC (2003)

[38] Riedweg, S., Pinchinat, S.: Maximally permissive controllers in all contexts. In: Workshop on Discrete Event Systems. Reims, France (2004)

[39] Rosu, G., Havelund, K.: Synthesizing dynamic programming algorithms from linear temporal logic formulae. Tech. rep. (2001)

[40] Saidi, H.: Towards automatic synthesis of security protocols (2002). URL http://www.csl.sri.com/papers/aaai_saidi02/

[41] Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3**(1), 30–50 (2000)

[42] Simpson, A.K.: Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS. In: Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, pp. 420–430. IEEE Computer Society Press (1995)

[43] Stirling, C.: Bisimulation, model checking, and other games (1997). Mathfit instructional meeting on games and computation

[44] Street, R.S., Emerson, E.A.: An automata theoretic procedure for the propositional $\mu$-calculus. Information and Computation **81**(3), 249–264 (1989)

[45] Ulidowski, I., Yuen, S.: Extending process languages with time. In: AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology. Springer-Verlag, London, UK (1997)

[46] Vardi, M.Y., Stockmeyer, L.: Improved upper and lower bounds for modal logics of programs. In: STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing, pp. 240–251. ACM Press, New York, NY, USA (1985). DOI http://doi.acm.org/10.1145/22145.22173

[47] Walukiewicz, I.: A complete deductive system for the $\mu$-calculus. Ph.D. thesis, Institute of Informatics, Warsaw University (1993)

[48] Wong-Toi, H., Dill, D.L.: Synthesizing processes and schedulers from temporal specifications. In: E.M. Clarke, R.P. Kurshan (eds.) CAV, *Lecture Notes in Computer Science*, vol. 531, pp. 272–281. Springer (1990)